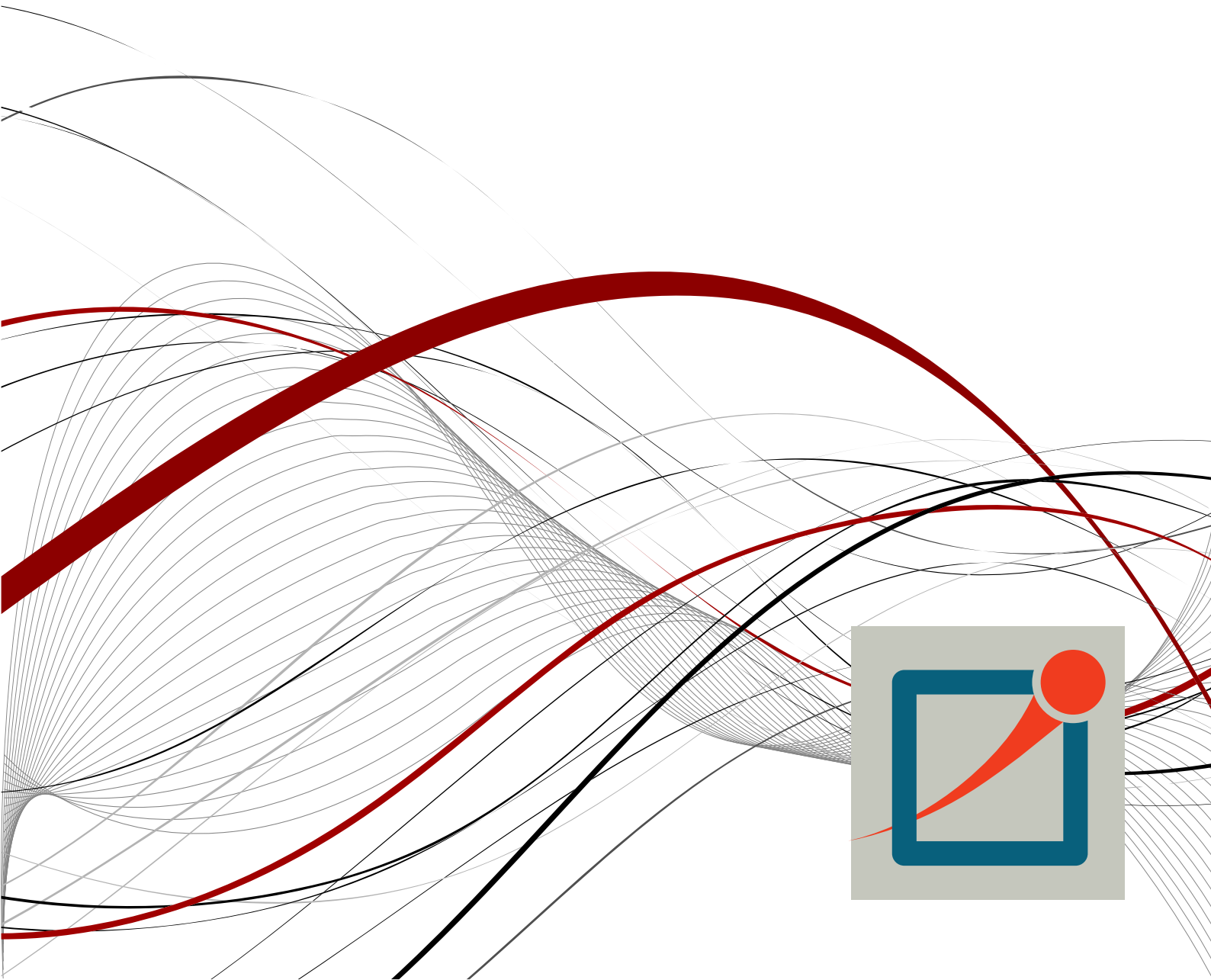
OMNEST

Simulation Manual

Version 6.x



Chapters

Contents

Chapter 1

Introduction

1.1 What Is OMNEST?

OMNEST is an object-oriented modular discrete event network simulation framework. It has a generic architecture, so it can be (and has been) used in various problem domains:

- modeling of wired and wireless communication networks
- protocol modeling
- modeling of queueing networks
- modeling of multiprocessors and other distributed hardware systems
- validating of hardware architectures
- evaluating performance aspects of complex software systems
- in general, modeling and simulation of any system where the discrete event approach is suitable, and can be conveniently mapped into entities communicating by exchanging messages.

OMNEST itself is not a simulator of anything concrete, but rather provides infrastructure and tools for *writing* simulations. One of the fundamental ingredients of this infrastructure is a component architecture for simulation models. Models are assembled from reusable components termed *modules*. Well-written modules are truly reusable, and can be combined in various ways like LEGO blocks.

Modules can be connected with each other via gates (other systems would call them ports), and combined to form compound modules. The depth of module nesting is not limited. Modules communicate through message passing, where messages may carry arbitrary data structures. Modules can pass messages along predefined paths via gates and connections, or directly to their destination; the latter is useful for wireless simulations, for example. Modules may have parameters that can be used to customize module behavior and/or to parameterize the model's topology. Modules at the lowest level of the module hierarchy are called simple modules, and they encapsulate model behavior. Simple modules are programmed in C++, and make use of the simulation library.

OMNEST simulations can be run under various user interfaces. Graphical, animating user interfaces are highly useful for demonstration and debugging purposes, and command-line user interfaces are best for batch execution.

The simulator as well as user interfaces and tools are highly portable. They are tested on the most common operating systems (Linux, Mac OS/X, Windows), and they can be compiled out of the box or after trivial modifications on most Unix-like operating systems.

OMNEST also supports parallel distributed simulation. OMNEST can use several mechanisms for communication between partitions of a parallel distributed simulation, for example MPI or named pipes. The parallel simulation algorithm can easily be extended, or new ones can be plugged in. Models do not need any special instrumentation to be run in parallel – it is just a matter of configuration. OMNEST can even be used for classroom presentation of parallel simulation algorithms, because simulations can be run in parallel even under the GUI that provides detailed feedback on what is going on.

OMNEST is the commercially supported version of OMNeT++. OMNeT++ is free only for academic and non-profit use; for commercial purposes, one needs to obtain OMNEST licenses from Simulcraft Inc.

1.2 Organization of This Manual

The manual is organized as follows:

- The Chapters 1 and 2 contain introductory material.
- The second group of chapters, 3, 4 and 7 are the programming guide. They present the NED language, describe the simulation concepts and their implementation in OMNEST, explain how to write simple modules, and describe the class library.
- The chapters 8 and 14explain how to customize the network graphics and how to write NED source code comments from which documentation can be generated.
- Chapters 9, 10, 11 and 12 deal with practical issues like building and running simulations and analyzing results, and describe the tools OMNEST provides to support these tasks.
- Chapter 16 is devoted to the support of distributed execution.
- Chapters 17 and 18 explain the architecture and internals of OMNEST, as well as ways to extend it and embed it into larger applications.
- The appendices provide a reference on the NED language, configuration options, file formats, and other details.

Chapter 2

Overview

2.1 Modeling Concepts

An OMNEST model consists of modules that communicate with message passing. The active modules are termed *simple modules*; they are written in C++, using the simulation class library. Simple modules can be grouped into *compound modules* and so forth; the number of hierarchy levels is unlimited. The whole model, called network in OMNEST, is itself a compound module. Messages can be sent either via connections that span modules or directly to other modules. The concept of simple and compound modules is similar to DEVS atomic and coupled models.

In Fig. 2.1, boxes represent simple modules (gray background) and compound modules. Arrows connecting small boxes represent connections and gates.

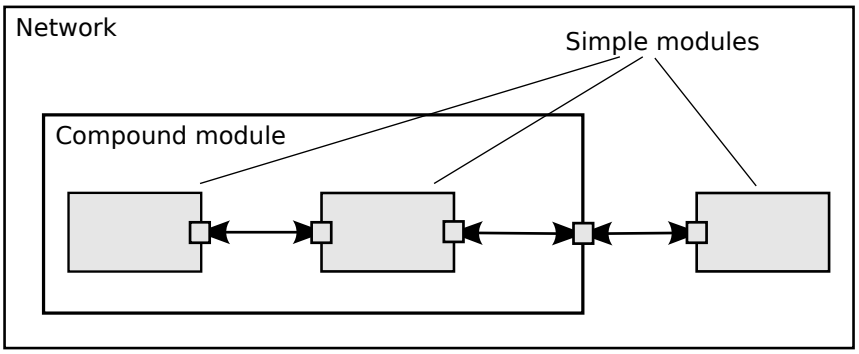


Figure 2.1: Simple and compound modules

Modules communicate with messages that may contain arbitrary data, in addition to usual attributes such as a timestamp. Simple modules typically send messages via gates, but it is also possible to send them directly to their destination modules. Gates are the input and output interfaces of modules: messages are sent through output gates and arrive through input gates. An input gate and output gate can be linked by a connection. Connections are created within a single level of module hierarchy; within a compound module, corresponding gates of two submodules, or a gate of one submodule and a gate of the compound module can be connected. Connections spanning hierarchy levels are not permitted, as they would

hinder model reuse. Because of the hierarchical structure of the model, messages typically travel through a chain of connections, starting and arriving in simple modules. Compound modules act like "cardboard boxes" in the model, transparently relaying messages between their inner realm and the outside world. Parameters such as propagation delay, data rate and bit error rate, can be assigned to connections. One can also define connection types with specific properties (termed channels) and reuse them in several places. Modules can have parameters. Parameters are used mainly to pass configuration data to simple modules, and to help define model topology. Parameters can take string, numeric, or boolean values. Because parameters are represented as objects in the program, parameters – in addition to holding constants – may transparently act as sources of random numbers, with the actual distributions provided with the model configuration. They may interactively prompt the user for the value, and they might also hold expressions referencing other parameters. Compound modules may pass parameters or expressions of parameters to their submodules.

OMNEST provides efficient tools for the user to describe the structure of the actual system. Some of the main features are the following:

- hierarchically nested modules
- modules are instances of module types
- modules communicate with messages through channels
- flexible module parameters
- topology description language

2.1.1 Hierarchical Modules

An OMNEST model consists of hierarchically nested modules that communicate by passing messages to each other. OMNEST models are often referred to as *networks*. The top level module is the *system module*. The system module contains *submodules* that can also contain submodules themselves (Fig. 2.1). The depth of module nesting is unlimited, allowing the user to reflect the logical structure of the actual system in the model structure.

Model structure is described in OMNEST's NED language.

Modules that contain submodules are termed *compound modules*, as opposed to *simple modules* at the lowest level of the module hierarchy. Simple modules contain the algorithms of the model. The user implements the simple modules in C++, using the OMNEST simulation class library.

2.1.2 Module Types

Both simple and compound modules are instances of *module types*. In describing the model, the user defines module types; instances of these module types serve as components for more complex module types. Finally, the user creates the system module as an instance of a previously defined module type; all modules of the network are instantiated as submodules and sub-submodules of the system module.

When a module type is used as a building block, it makes no difference whether it is a simple or compound module. This allows the user to split a simple module into several simple modules embedded into a compound module, or vice versa, to aggregate the functionality of a

compound module into a single simple module, without affecting existing users of the module type.

Module types can be stored in files separately from the place of their actual usage. This means that the user can group existing module types and create *component libraries*. This feature will be discussed later, in chapter 11.

2.1.3 Messages, Gates, Links

Modules communicate by exchanging *messages*. In an actual simulation, messages can represent frames or packets in a computer network, jobs or customers in a queuing network or other types of mobile entities. Messages can contain arbitrarily complex data structures. Simple modules can send messages either directly to their destination or along a predefined path, through gates and connections.

The "local simulation time" of a module advances when the module receives a message. The message can arrive from another module or from the same module (*self-messages* are used to implement timers).

Gates are the input and output interfaces of modules; messages are sent out through output gates and arrive through input gates.

Each *connection* (also called *link*) is created within a single level of the module hierarchy: within a compound module, one can connect the corresponding gates of two submodules, or a gate of one submodule and a gate of the compound module (Fig. 2.1).

Because of the hierarchical structure of the model, messages typically travel through a series of connections, starting and arriving in simple modules. Compound modules act like "cardboard boxes" in the model, transparently relaying messages between their inner realm and the outside world.

2.1.4 Modeling of Packet Transmissions

To facilitate the modeling of communication networks, connections can be used to model physical links. Connections support the following parameters: *data rate*, *propagation delay*, *bit error rate* and *packet error rate*, and may be disabled. These parameters and the underlying algorithms are encapsulated into *channel* objects. The user can parameterize the channel types provided by OMNEST, and also create new ones.

When data rates are in use, a packet object is by default delivered to the target module at the simulation time that corresponds to the end of the packet reception. Since this behavior is not suitable for the modeling of some protocols (e.g. half-duplex Ethernet), OMNEST provides the possibility for the target module to specify that it wants the packet object to be delivered to it when the packet reception starts.

2.1.5 Parameters

Modules can have parameters. Parameters can be assigned in either the NED files or the configuration file `omnetpp.ini`.

Parameters can be used to customize simple module behavior, and to parameterize the model topology.

Parameters can take string, numeric or boolean values, or can contain XML data trees. Numeric values include expressions using other parameters and calling C functions, random variables from different distributions, and values input interactively by the user.

Numeric-valued parameters can be used to construct topologies in a flexible way. Within a compound module, parameters can define the number of submodules, number of gates, and the way the internal connections are made.

2.1.6 Topology Description Method

The user defines the structure of the model in NED language descriptions (Network Description). The NED language will be discussed in detail in chapter 3.

2.2 Programming the Algorithms

The simple modules of a model contain algorithms as C++ functions. The full flexibility and power of the programming language can be used, supported by the OMNEST simulation class library. The simulation programmer can choose between event-driven and process-style description, and freely use object-oriented concepts (inheritance, polymorphism etc) and design patterns to extend the functionality of the simulator.

Simulation objects (messages, modules, queues etc.) are represented by C++ classes. They have been designed to work together efficiently, creating a powerful simulation programming framework. The following classes are part of the simulation class library:

- module, gate, parameter, channel
- message, packet
- container classes (e.g. queue, array)
- data collection classes
- statistic and distribution estimation classes (histograms, P^2 algorithm for calculating quantiles etc.)

The classes are also specially instrumented, allowing one to traverse objects of a running simulation and display information about them such as name, class name, state variables or contents. This feature makes it possible to create a simulation GUI where all internals of the simulation are visible.

2.3 Using OMNEST

2.3.1 Building and Running Simulations

This section provides insights into working with OMNEST in practice. Issues such as model files and compiling and running simulations are discussed.

An OMNEST model consists of the following parts:

- NED language topology description(s) (`.ned` files) that describe the module structure with parameters, gates, etc. NED files can be written using any text editor, but the OMNEST IDE provides excellent support for two-way graphical and text editing.
- Message definitions (`.msg` files) that let one define message types and add data fields to them. OMNEST will translate message definitions into full-fledged C++ classes.
- Simple module sources. They are C++ files, with `.h`/`.cc` suffix.

The simulation system provides the following components:

- Simulation kernel. This contains the code that manages the simulation and the simulation class library. It is written in C++, compiled into a shared or static library.
- User interfaces. OMNEST user interfaces are used in simulation execution, to facilitate debugging, demonstration, or batch execution of simulations. They are written in C++, compiled into libraries.

Simulation programs are built from the above components. First, `.msg` files are translated into C++ code using the `opp_msgc`. program. Then all C++ sources are compiled and linked with the simulation kernel and a user interface library to form a simulation executable or shared library. NED files are loaded dynamically in their original text forms when the simulation program starts.

Running the Simulation and Analyzing the Results

The simulation may be compiled as a standalone program executable, or as a shared library to be run using OMNEST's `opp_run` utility. When the program is started, it first reads the NED files, then the configuration file usually called `omnetpp.ini`. The configuration file contains settings that control how the simulation is executed, values for model parameters, etc. The configuration file can also prescribe several simulation runs; in the simplest case, they will be executed by the simulation program one after another.

The output of the simulation is written into result files: output vector files, output scalar files, and possibly the user's own output files. OMNEST contains an Integrated Development Environment (IDE) that provides rich environment for analyzing these files. Output files are line-oriented text files which makes it possible to process them with a variety of tools and programming languages as well, including Matlab, GNU R, Perl, Python, and spreadsheet programs.

User Interfaces

The primary purpose of user interfaces is to make the internals of the model visible to the user, to control simulation execution, and possibly allow the user to intervene by changing variables/objects inside the model. This is very important in the development/debugging phase of the simulation project. Equally important, a hands-on experience allows the user to get a feel of the model's behavior. The graphical user interface can also be used to demonstrate a model's operation.

The same simulation model can be executed with various user interfaces, with no change in the model files themselves. The user would typically test and debug the simulation with a powerful graphical user interface, and finally run it with a simple, fast user interface that supports batch execution.

Component Libraries

Module types can be stored in files separate from the place of their actual use, enabling the user to group existing module types and create component libraries.

Universal Standalone Simulation Programs

A simulation executable can store several independent models that use the same set of simple modules. The user can specify in the configuration file which model is to be run. This allows one to build one large executable that contains several simulation models, and distribute it as a standalone simulation tool. The flexibility of the topology description language also supports this approach.

2.3.2 What Is in the Distribution

An OMNEST installation contains the following subdirectories. Depending on the platform, there may also be additional directories present, containing software bundled with OMNEST.)

The simulation system itself:

```
omnetpp/          OMNEST root directory
  bin/            OMNEST executables
  include/        header files for simulation models
  lib/            library files
  images/         icons and backgrounds for network graphics
  doc/            manuals, readme files, license, APIs, etc.
    ide-customization-guide/ how to write new wizards for the IDE
    ide-developersguide/ writing extensions for the IDE
    manual/       manual in HTML
    ned2/         DTD definition of the XML syntax for NED files
    tictoc-tutorial/  introduction into using OMNEST
    api/          API reference in HTML
    nedxml-api/   API reference for the NEDXML library
    parsim-api/   API reference for the parallel simulation library
  src/            OMNEST sources
    sim/          simulation kernel
      parsim/     files for distributed execution
      netbuilder/files for dynamically reading NED files
    envir/        common code for user interfaces
    cmdenv/       command-line user interface
    qtenv/        Qt-based user interface
    nedxml/       NEDXML library, opp_nedtool, opp_msgtool
    scave/        result analysis library, opp_scavetool
    eventlog/     eventlog processing library
    layout/       graph layouter for network graphics
    common/       common library
    utils/        opp_makemake, opp_test, etc.
  ide/            Simulation IDE
  python/         Python libraries for OMNEST
    omnetpp/      Python package name
      scave/      Python API for result analysis
```

```
   ...
test/          Regression test suite
  core/        tests for the simulation library
  anim/        tests for graphics and animation
  dist/        tests for the built-in distributions
  makemake/    tests for opp_makemake
  ...
```

The Eclipse-based Simulation IDE is in the `ide` directory.

```
ide/           Simulation IDE
  features/    Eclipse feature definitions
  plugins/     IDE plugins (extensions to the IDE can be dropped here)
  ...
```

The Windows version of OMNEST contains a redistribution of the MinGW gcc compiler, together with a copy of MSYS that provides Unix tools commonly used in Makefiles. The MSYS directory also contains various 3rd party open-source libraries needed to compile and run OMNEST.

```
tools/       Platform specific tools and compilers (e.g. MinGW/MSYS on Windows)
```

Sample simulations are in the `samples` directory.

```
samples/     directories for sample simulations
  aloha/     models the Aloha protocol
  cqn/       Closed Queueing Network
  ...
```

The `contrib` directory contains material from the OMNEST community.

```
contrib/     directory for contributed material
  akaroa/    Patch to compile akaroa on newer gcc systems
  topologyexport/  Export the topology of a model in runtime
  ...
```

Chapter 3

The NED Language

3.1 NED Overview

The user describes the structure of a simulation model in the NED language. NED stands for Network Description. NED lets the user declare simple modules, and connect and assemble them into compound modules. The user can label some compound modules as *networks*; that is, self-contained simulation models. Channels are another component type, whose instances can also be used in compound modules.

The NED language has several features which let it scale well to large projects:

Hierarchical. The traditional way to deal with complexity is by introducing hierarchies. In OMNEST, any module which would be too complex as a single entity can be broken down into smaller modules, and used as a compound module.

Component-Based. Simple modules and compound modules are inherently reusable, which not only reduces code copying, but more importantly, allows component libraries (like the INET Framework, MiXiM, Castalia, etc.) to exist.

Interfaces. Module and channel interfaces can be used as a placeholder where normally a module or channel type would be used, and the concrete module or channel type is determined at network setup time by a parameter. Concrete module types have to "implement" the interface they can substitute. For example, given a compound module type named `MobileHost` contains a `mobility` submodule of the type `IMobility` (where `IMobility` is a module interface), the actual type of `mobility` may be chosen from the module types that implemented `IMobility` (`RandomWalkMobility`, `TurtleMobility`, etc.)

Inheritance. Modules and channels can be subclassed. Derived modules and channels may add new parameters, gates, and (in the case of compound modules) new submodules and connections. They may set existing parameters to a specific value, and also set the gate size of a gate vector. This makes it possible, for example, to take a `GenericTCPClientApp` module and derive an `FTPClientApp` from it by setting certain parameters to a fixed value; or to derive a `WebClientHost` compound module from a `BaseHost` compound module by adding a `WebClientApp` submodule and connecting it to the inherited `TCP` submodule.

Packages. The NED language features a Java-like package structure, to reduce the risk of

name clashes between different models. NEDPATH (similar to Java's CLASSPATH) has also been introduced to make it easier to specify dependencies among simulation models.

Inner types. Channel types and module types used locally by a compound module can be defined within the compound module, in order to reduce namespace pollution.

Metadata annotations. It is possible to annotate module or channel types, parameters, gates and submodules by adding properties. Metadata are not used by the simulation kernel directly, but they can carry extra information for various tools, the runtime environment, or even for other modules in the model. For example, a module's graphical representation (icon, etc) or the prompt string and measurement unit (milliwatt, etc) of a parameter are already specified as metadata annotations.

NOTE: The NED language has changed significantly in the 4.0 version. Inheritance, interfaces, packages, inner types, metadata annotations, inout gates were all added in the 4.0 release, together with many other features. Since the basic syntax has changed as well, old NED files need to be converted to the new syntax. There are automated tools for this purpose, so manual editing is only needed to take advantage of new NED features.

The NED language has an equivalent tree representation which can be serialized to XML; that is, NED files can be converted to XML and back without loss of data, including comments. This lowers the barrier for programmatic manipulation of NED files; for example extracting information, refactoring and transforming NED, generating NED from information stored in other systems like SQL databases, and so on.

NOTE: This chapter is going to explain the NED language gradually, via examples. A more formal and concise treatment can be found in Appendix B.

3.2 NED Quickstart

In this section we introduce the NED language via a complete and reasonably real-life example: a communication network.

Our hypothetical network consists of nodes. On each node there is an application running which generates packets at random intervals. The nodes are routers themselves as well. We assume that the application uses datagram-based communication, so that we can leave out the transport layer from the model.

3.2.1 The Network

First we'll define the network, then in the next sections we'll continue to define the network nodes.

Let the network topology be as in Figure 3.1.

The corresponding NED description would look like this:

```
//
// A network
//
network Network
{
```

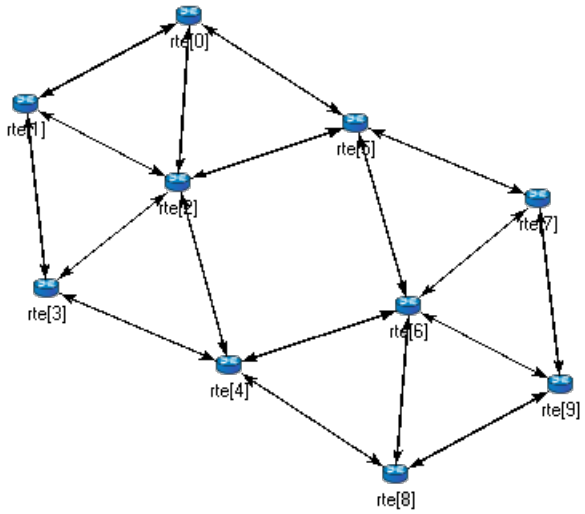


Figure 3.1: The network

```
    submodules:
        node1: Node;
        node2: Node;
        node3: Node;
        ...
    connections:
        node1.port++ <--> {datarate=100Mbps;} <--> node2.port++;
        node2.port++ <--> {datarate=100Mbps;} <--> node4.port++;
        node4.port++ <--> {datarate=100Mbps;} <--> node6.port++;
        ...
}
```

The above code defines a network type named `Network`. Note that the NED language uses the familiar curly brace syntax, and "`//`" to denote comments.

> **NOTE:** Comments in NED not only make the source code more readable, but in the OMNEST IDE they also are displayed at various places (tooltips, content assist, etc), and become part of the documentation extracted from the NED files. The NED documentation system, not unlike *JavaDoc* or *Doxygen*, will be described in Chapter 14.

The network contains several nodes, named `node1`, `node2`, etc. from the NED module type `Node`. We'll define `Node` in the next sections.

The second half of the declaration defines how the nodes are to be connected. The double arrow means bidirectional connection. The connection points of modules are called gates, and the `port++` notation adds a new gate to the `port[]` gate vector. Gates and connections will be covered in more detail in sections 3.7 and 3.9. Nodes are connected with a channel that has a data rate of 100Mbps.

> **NOTE:** In many other systems, the equivalent of OMNEST gates are called *ports*. We have retained the term *gate* to reduce collisions with other uses of the otherwise overloaded word *port*: router port, TCP port, I/O port, etc.

The above code would be placed into a file named `Net6.ned`. It is a convention to put every NED definition into its own file and to name the file accordingly, but it is not mandatory to do so.

One can define any number of networks in the NED files, and for every simulation the user has to specify which network to set up. The usual way of specifying the network is to put the **`network`** option into the configuration (by default the `omnetpp.ini` file):

```
[General]
network = Network
```

3.2.2 Introducing a Channel

It is cumbersome to have to repeat the data rate for every connection. Luckily, NED provides a convenient solution: one can create a new channel type that encapsulates the data rate setting, and this channel type can be defined inside the network so that it does not litter the global namespace.

The improved network will look like this:

```
//
// A Network
//
network Network
{
    types:
        channel C extends ned.DatarateChannel {
            datarate = 100Mbps;
        }
    submodules:
        node1: Node;
        node2: Node;
        node3: Node;
        ...
    connections:
        node1.port++ <--> C <--> node2.port++;
        node2.port++ <--> C <--> node4.port++;
        node4.port++ <--> C <--> node6.port++;
        ...
}
```

Later sections will cover the concepts used (inner types, channels, the `DatarateChannel` built-in type, inheritance) in detail.

3.2.3 The App, Routing, and Queue Simple Modules

Simple modules are the basic building blocks for other (compound) modules, denoted by the **`simple`** keyword. All active behavior in the model is encapsulated in **`simple`** modules. Behavior is defined with a C++ class; NED files only declare the externally visible interface of the module (gates, parameters).

In our example, we could define `Node` as a simple module. However, its functionality is quite complex (traffic generation, routing, etc), so it is better to implement it with several smaller

simple module types which we are going to assemble into a compound module. We'll have one simple module for traffic generation (`App`), one for routing (`Routing`), and one for queueing up packets to be sent out (`Queue`). For brevity, we omit the bodies of the latter two in the code below.

```
simple App
{
    parameters:
        int destAddress;
        ...
        @display("i=block/browser");
    gates:
        input in;
        output out;
}

simple Routing
{
    ...
}

simple Queue
{
    ...
}
```

By convention, the above simple module declarations go into the `App.ned`, `Routing.ned` and `Queue.ned` files.

> **NOTE:** Note that module type names (`App`, `Routing`, `Queue`) begin with a capital letter, and parameter and gate names begin with lowercase – this is the recommended naming convention. Capitalization matters because the language is case sensitive.

Let us look at the first simple module type declaration. `App` has a parameter called `destAddress` (others have been omitted for now), and two gates named `out` and `in` for sending and receiving application packets.

The argument of `@display()` is called a *display string*, and it defines the rendering of the module in graphical environments; `"i=..."` defines the default icon.

Generally, `@`-words like `@display` are called *properties* in NED, and they are used to annotate various objects with metadata. Properties can be attached to files, modules, parameters, gates, connections, and other objects, and parameter values have a very flexible syntax.

3.2.4 The Node Compound Module

Now we can assemble `App`, `Routing` and `Queue` into the compound module `Node`. A compound module can be thought of as a "cardboard box" that groups other modules into a larger unit, which can further be used as a building block for other modules; networks are also a kind of compound module.

```
module Node
{
```

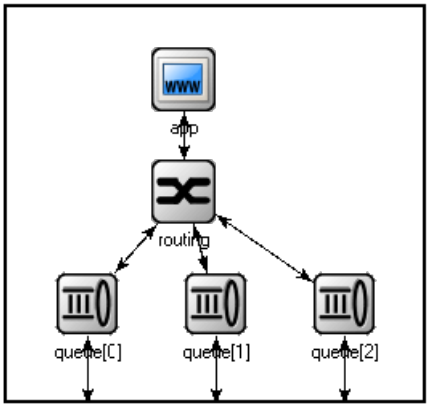


Figure 3.2: The Node compound module

```
    parameters:
        int address;
        @display("i=misc/node_vs,gold");
    gates:
        inout port[];
    submodules:
        app: App;
        routing: Routing;
        queue[sizeof(port)]: Queue;
    connections:
        routing.localOut --> app.in;
        routing.localIn <-- app.out;
        for i=0..sizeof(port)-1 {
            routing.out[i] --> queue[i].in;
            routing.in[i] <-- queue[i].out;
            queue[i].line <--> port[i];
        }
}
```

Compound modules, like simple modules, may have parameters and gates. Our `Node` module contains an `address` parameter, plus a *gate vector* of unspecified size, named `port`. The actual gate vector size will be determined implicitly by the number of neighbours when we create a network from nodes of this type. The type of `port[]` is `inout`, which allows bidirectional connections.

The modules that make up the compound module are listed under **`submodules`**. Our `Node` compound module type has an `app` and a `routing` *submodule*, plus a `queue[]` *submodule vector* that contains one `Queue` module for each port, as specified by `[sizeof(port)]`. (It is legal to refer to `[sizeof(port)]` because the network is built in top-down order, and the node is already created and connected at network level when its submodule structure is built out.)

In the **`connections`** section, the submodules are connected to each other and to the parent module. Single arrows are used to connect input and output gates, and double arrows connect inout gates, and a **`for`** loop is utilized to connect the `routing` module to each `queue` module, and to connect the outgoing/incoming link (`line` gate) of each queue to the corresponding

port of the enclosing module.

3.2.5 Putting It Together

We have created the NED definitions for this example, but how are they used by OMNEST? When the simulation program is started, it loads the NED files. The program should already contain the C++ classes that implement the needed simple modules, `App`, `Routing` and `Queue`; their C++ code is either part of the executable or is loaded from a shared library. The simulation program also loads the configuration (`omnetpp.ini`), and determines from it that the simulation model to be run is the `Network` network. Then the network is instantiated for simulation.

The simulation model is built in a top-down preorder fashion. This means that starting from an empty system module, all submodules are created, their parameters and gate vector sizes are assigned, and they are fully connected before the submodule internals are built.

* * *

In the following sections we'll go through the elements of the NED language and look at them in more detail.

3.3 Simple Modules

Simple modules are the active components in the model. Simple modules are defined with the **`simple`** keyword.

An example simple module:

```
simple Queue
{
    parameters:
        int capacity;
        @display("i=block/queue");
    gates:
        input in;
        output out;
}
```

Both the **`parameters`** and **`gates`** sections are optional, that is, they can be left out if there is no parameter or gate. In addition, the **`parameters`** keyword itself is optional too; it can be left out even if there are parameters or properties.

Note that the NED definition doesn't contain any code to define the operation of the module: that part is expressed in C++. By default, OMNEST looks for C++ classes of the same name as the NED type (so here, `Queue`).

One can explicitly specify the C++ class with the `@class` property. Classes with namespace qualifiers are also accepted, as shown in the following example that uses the `mylib::Queue` class:

```
simple Queue
{
    parameters:
        int capacity;
        @class(mylib::Queue);
        @display("i=block/queue");
    gates:
        input in;
        output out;
}
```

If there are several modules whose C++ implementation classes are in the same namespace, a better alternative to `@class` is the `@namespace` property. The C++ namespace given with `@namespace` will be prepended to the normal class name. In the following example, the C++ classes will be `mylib::App`, `mylib::Router` and `mylib::Queue`:

```
@namespace(mylib);

simple App {
   ...
}

simple Router {
   ...
}

simple Queue {
   ...
}
```

The `@namespace` property may not only be specified at file level as in the above example, but for packages as well. When placed in a file called `package.ned`, the namespace will apply to all components in that package and below.

The implementation C++ classes need to be subclassed from the `cSimpleModule` library class; chapter 4 of this manual describes in detail how to write them.

Simple modules can be extended (or specialized) via subclassing. The motivation for subclassing can be to set some open parameters or gate sizes to a fixed value (see 3.6 and 3.7), or to replace the C++ class with a different one. Now, by default, the derived NED module type will *inherit* the C++ class from its base, so it is important to remember that you need to write out `@class` if you want it to use the new class.

The following example shows how to specialize a module by setting a parameter to a fixed value (and leaving the C++ class unchanged):

```
simple Queue
{
   int capacity;
   ...
}

simple BoundedQueue extends Queue
{
   capacity = 10;
```

```
}
```

In the next example, the author wrote a `PriorityQueue` C++ class, and wants to have a corresponding NED type, derived from `Queue`. However, it does not work as expected:

```
simple PriorityQueue extends Queue // wrong! still uses the Queue C++ class
{
}
```

The correct solution is to add a `@class` property to override the inherited C++ class:

```
simple PriorityQueue extends Queue
{
    @class(PriorityQueue);
}
```

Inheritance in general will be discussed in section 3.13.

3.4 Compound Modules

A compound module groups other modules into a larger unit. A compound module may have gates and parameters like a simple module, but no active behavior is associated with it.[1]

> **NOTE:** When there is a temptation to add code to a compound module, then encapsulate the code into a simple module, and add it as a submodule.

A compound module declaration may contain several sections, all of them optional:

```
module Host
{
    types:
        ...
    parameters:
        ...
    gates:
        ...
    submodules:
        ...
    connections:
        ...
}
```

Modules contained in a compound module are called submodules, and they are listed in the `submodules` section. One can create arrays of submodules (i.e. submodule vectors), and the submodule type may come from a parameter.

Connections are listed under the `connections` section of the declaration. One can create connections using simple programming constructs (loop, conditional). Connection behaviour can be defined by associating a channel with the connection; the channel type may also come from a parameter.

[1] Although the C++ class for a compound module can be overridden with the `@class` property, this is a feature that should probably never be used. Encapsulate the code into a simple module, and add it as a submodule.

Module and channel types only used locally can be defined in the `types` section as inner types, so that they do not pollute the namespace.

Compound modules may be extended via subclassing. Inheritance may add new submodules and new connections as well, not only parameters and gates. Also, one may refer to inherited submodules, to inherited types etc. What is not possible is to "de-inherit" or modify submodules or connections.

In the following example, we show how to assemble common protocols into a "stub" for wireless hosts, and add user agents via subclassing.[2]

```
module WirelessHostBase
{
    gates:
        input radioIn;
    submodules:
        tcp: TCP;
        ip: IP;
        wlan: Ieee80211;
    connections:
        tcp.ipOut --> ip.tcpIn;
        tcp.ipIn <-- ip.tcpOut;
        ip.nicOut++ --> wlan.ipIn;
        ip.nicIn++ <-- wlan.ipOut;
        wlan.radioIn <-- radioIn;
}

module WirelessHost extends WirelessHostBase
{
    submodules:
        webAgent: WebAgent;
    connections:
        webAgent.tcpOut --> tcp.appIn++;
        webAgent.tcpIn <-- tcp.appOut++;
}
```

The `WirelessHost` compound module can further be extended, for example with an Ethernet port:

```
module DesktopHost extends WirelessHost
{
    gates:
        inout ethg;
    submodules:
        eth: EthernetNic;
    connections:
        ip.nicOut++ --> eth.ipIn;
        ip.nicIn++ <-- eth.ipOut;
        eth.phy <--> ethg;
}
```

[2] Module types, gate names, etc. used in the example are fictional, not based on an actual OMNEST-based model framework

3.5 Channels

Channels encapsulate parameters and behaviour associated with connections. Channels are like simple modules, in the sense that there are C++ classes behind them. The rules for finding the C++ class for a NED channel type is the same as with simple modules: the default class name is the NED type name unless there is a `@class` property (`@namespace` is also recognized), and the C++ class is inherited when the channel is subclassed.

Thus, the following channel type would expect a `CustomChannel` C++ class to be present:

```
channel CustomChannel  // requires a CustomChannel C++ class
{
}
```

The practical difference compared to modules is that one rarely needs to write custom channel C++ class because there are predefined channel types that one can subclass from, inheriting their C++ code. The predefined types are: `ned.IdealChannel`, `ned.DelayChannel` and `ned.DatarateChannel`. ("`ned`" is the package name; one can get rid of it by importing the types with the `import ned.*` directive. Packages and imports are described in section 3.14.)

`IdealChannel` has no parameters, and lets through all messages without delay or any side effect. A connection without a channel object and a connection with an `IdealChannel` behave in the same way. Still, `IdealChannel` has its uses, for example when a channel object is required so that it can carry a new property or parameter that is going to be read by other parts of the simulation model.

`DelayChannel` has two parameters:

- `delay` is a `double` parameter which represents the propagation delay of the message. Values need to be specified together with a time unit (`s`, `ms`, `us`, etc.)
- `disabled` is a boolean parameter that defaults to `false`; when set to `true`, the channel object will drop all messages.

`DatarateChannel` has a few additional parameters compared to `DelayChannel`:

- `datarate` is a `double` parameter that represents the data rate of the channel. Values need to be specified in bits per second or its multiples as unit (`bps`, `kbps`, `Mbps`, `Gbps`, etc.) Zero is treated specially and results in zero transmission duration, i.e. it stands for infinite bandwidth. Zero is also the default. Data rate is used for calculating the transmission duration of packets.
- `ber` and `per` stand for Bit Error Rate and Packet Error Rate, and allow basic error modelling. They expect a `double` in the $[0,1]$ range. When the channel decides (based on random numbers) that an error occurred during transmission of a packet, it sets an error flag in the packet object. The receiver module is expected to check the flag, and discard the packet as corrupted if it is set. The default `ber` and `per` are zero.

NOTE: There is no channel parameter that specifies whether the channel delivers the message object to the destination module at the end or at the start of the reception; that is decided by the C++ code of the target simple module. See the `setDeliverOnReceptionStart()` method of `cGate`.

The following example shows how to create a new channel type by specializing `DatarateChannel`:

```
channel Ethernet100 extends ned.DatarateChannel
{
    datarate = 100Mbps;
    delay = 100us;
    ber = 1e-10;
}
```

NOTE: The three built-in channel types are also used for connections where the channel type is not explicitly specified.

One may add parameters and properties to channels via subclassing, and may modify existing ones. In the following example, we introduce distance-based calculation of the propagation delay:

```
channel DatarateChannel2 extends ned.DatarateChannel
{
    double distance @unit(m);
    delay = this.distance / 200000km * 1s;
}
```

Parameters are primarily intended to be read by the underlying C++ class, but new parameters may also be added as annotations to be used by other parts of the model. For example, a `cost` parameter may be used for routing decisions in routing module, as shown in the example below. The example also shows annotation using properties (`@backbone`).

```
channel Backbone extends ned.DatarateChannel
{
    @backbone;
    double cost = default(1);
}
```

3.6 Parameters

Parameters are variables that belong to a module. Parameters can be used in building the topology (number of nodes, etc), and to supply input to C++ code that implements simple modules and channels.

Parameters can be of type **double**, **int**, **bool**, **string**, **xml** and **object**; they can also be declared **volatile**. For the numeric types, a unit of measurement can also be specified (`@unit` property).

Parameters can get their value from NED files or from the configuration (`omnetpp.ini`). A default value can also be given (`default(...)`), which is used if the parameter is not assigned otherwise.

The following example shows a simple module that has five parameters, three of which have default values:

```
simple App
{
    parameters:
        string protocol;       // protocol to use: "UDP" / "IP" / "ICMP" / ...
```

```
        int destAddress;       // destination address
        volatile double sendInterval @unit(s) = default(exponential(1s));
                               // time between generating packets
        volatile int packetLength @unit(byte) = default(100B);
                               // length of one packet
        volatile int timeToLive = default(32);
                               // maximum number of network hops to survive
    gates:
        input in;
        output out;
}
```

3.6.1 Assigning a Value

Parameters may get their values in several ways: from NED code, from the configuration (`omnetpp.ini`), or even, interactively from the user. NED lets one assign parameters at several places: in subclasses via inheritance; in submodule and connection definitions where the NED type is instantiated; and in networks and compound modules that directly or indirectly contain the corresponding submodule or connection.

For instance, one could specialize the above `App` module type via inheritance with the following definition:

```
simple PingApp extends App
{
    parameters:
        protocol = "ICMP/ECHO"
        sendInterval = default(1s);
        packetLength = default(64byte);
}
```

This definition sets the `protocol` parameter to a fixed value (`"ICMP/ECHO"`), and changes the default values of the `sendInterval` and `packetLength` parameters. `protocol` is now locked down in `PingApp`, its value cannot be modified via further subclassing or other ways. `sendInterval` and `packetLength` are still unassigned here, only their default values have been overwritten.

Now, let us see the definition of a `Host` compound module that uses `PingApp` as submodule:

```
module Host
{
    submodules:
        ping : PingApp {
            packetLength = 128B; // always ping with 128-byte packets
        }
        ...
}
```

This definition sets the `packetLength` parameter to a fixed value. It is now hardcoded that `Host`s send 128-byte ping packets; this setting cannot be changed from NED or the configuration.

It is not only possible to set a parameter from the compound module that contains the submodule, but also from modules higher up in the module tree. A network that employs several

`Host` modules could be defined like this:

```
network Network
{
    submodules:
        host[100]: Host {
            ping.timeToLive = default(3);
            ping.destAddress = default(0);
        }
        ...
}
```

Parameter assignment can also be placed into the `parameters` block of the parent compound module, which provides additional flexibility. The following definition sets up the hosts so that half of them pings host #50, and the other half pings host #0:

```
network Network
{
    parameters:
        host[*].ping.timeToLive = default(3);
        host[0..49].ping.destAddress = default(50);
        host[50..].ping.destAddress = default(0);

    submodules:
        host[100]: Host;
        ...
}
```

Note the use of asterisk to match any index, and `..` to match index ranges.

If there were a number of individual hosts instead of a submodule vector, the network definition could look like this:

```
network Network
{
    parameters:
        host*.ping.timeToLive = default(3);
        host{0..49}.ping.destAddress = default(50);
        host{50..}.ping.destAddress = default(0);

    submodules:
        host0: Host;
        host1: Host;
        host2: Host;
        ...
        host99: Host;
}
```

An asterisk matches any substring not containing a dot, and a `..` within a pair of curly braces matches a natural number embedded in a string.

In most assigments we have seen above, the left hand side of the equal sign contained a dot and often a wildcard as well (asterisk or numeric range); we call these assignments *pattern assignments* or *deep assignments*.

There is one more wildcard that can be used in pattern assignments, and this is the double

asterisk; it matches any sequence of characters including dots, so it can match multiple path elements. An example:

```
network Network
{
    parameters:
        **.timeToLive = default(3);
        **.destAddress = default(0);
    submodules:
        host0: Host;
        host1: Host;
        ...
}
```

Note that some assignments in the above examples changed default values, while others set parameters to fixed values. Parameters that received no fixed value in the NED files can be assigned from the configuration (`omnetpp.ini`).

> **IMPORTANT:** A non-default value assigned from NED cannot be overwritten later in NED or from ini files; it becomes "hardcoded" as far as ini files and NED usage are concerned.

A parameter can be assigned in the configuration using a similar syntax as NED pattern assignments (actually, it would be more historically accurate to say it the other way round, that NED pattern assignments use a similar syntax to ini files):

```
Network.host[*].ping.sendInterval = 500ms  # for the host[100] example
Network.host*.ping.sendInterval = 500ms    # for the host0,host1,... example
**.sendInterval = 500ms
```

One often uses the double asterisk to save typing. One can write

```
**.ping.sendInterval = 500ms
```

Or if one is certain that only ping modules have `sendInterval` parameters, the following will suffice:

```
**.sendInterval = 500ms
```

Parameter assignments in the configuration are described in section 10.3.

One can also write expressions, including stochastic expressions, in NED files and in ini files as well. For example, here's how one can add jitter to the sending of ping packets:

```
**.sendInterval = 1s + normal(0s, 0.001s)  # or just: normal(1s, 0.001s)
```

If there is no assignment for a parameter in NED or in the ini file, the default value (given with `=default(...)` in NED) will be applied implicitly. If there is no default value, the user will be asked, provided the simulation program is allowed to do that; otherwise there will be an error. (Interactive mode is typically disabled for batch executions where it would do more harm than good.)

It is also possible to explicitly apply the default (this can sometimes be useful):

```
**.sendInterval = default
```

Finally, one can explicitly ask the simulator to prompt the user interactively for the value (again, provided that interactivity is enabled; otherwise this will result in an error):

```
**.sendInterval = ask
```

NOTE: How can one decide whether to assign a parameter from NED or from an ini file? The advantage of ini files is that they allow a cleaner separation of the *model* and *experiments*. NED files (together with C++ code) are considered to be part of the model, and to be more or less constant. Ini files, on the other hand, are for experimenting with the model by running it several times with different parameters. Thus, parameters that are expected to change (or make sense to be changed) during experimentation should be put into ini files.

3.6.2 Expressions

Parameter values may be given with expressions. NED language expressions have a C-like syntax, with additions like quantities (numbers with measurement units, e.g. `100Gbps`) and JSON constructs. Compared to C, there are some variations on operator names: binary and logical XOR are `#` and `##`, while `^` has been reassigned to *power-of* instead. The `+` operator does string concatenation as well as numeric addition. There are two extra operators: `<=>` ("spaceship") and `=~` (string match). The JSON constructs are the *array* and the *object* syntaxes, which will be covered in section 3.6.8. Keyword constants include **`true`**, **`false`**, **`nan`** (floating point Not-a-Number), **`inf`** (infinity), **`null`** and its synonym **`nullptr`**, and also **`undefined`** which represents the missing value.

The spaceship operator `<=>` compares its two arguments and returns the result ("less", "equal", "greater" and "not applicable") in the form of a negative, zero, positive or `nan` double number, respectively.

```
2 <=> 2  // --> 0
10 <=> 5  // --> 1
2 <=> nan // --> nan
```

The string match operator `=~` is used as *string* `=~` *pattern*, and returns a boolean that indicates whether if the second argument (the pattern) matches the first one (the string). Pattern syntax and rules are similar to those used in `omnetpp.ini` files: case sensitive, full-string match, where an asterisk `*` matches zero or more of any character except dot, a double asterisk `**` matches zero or more characters (including dot), and notations also exist to express embedded numbers and square-bracketed numeric indices within a numeric range.

```
"foo" =~ "f*" // --> true
"foo" =~ "b*" // --> false
"foo" =~ "F*" // --> false
"foo.bar.baz" =~ "*.baz" // --> false
"foo.bar.baz" =~ "**.baz" // --> true
"foo[15]" =~ "foo[5..20]" // --> true
"foo15" =~ "foo{5..20}" // --> true
```

Expressions may refer to module parameters, gate vector and module vector sizes (using the **`sizeof`** operator), existence of a submodule or submodule vector (**`exists`** operator), and the index of the current module in a submodule vector (**`index`**).

The special operator **`expr()`** can be used to pass a formula into a module as a parameter (3.6.9).

Expressions may also utilize various numeric, string, stochastic and miscellaneous other functions (`fabs()`, `uniform()`, `lognormal()`, `substring()`, `readFile()`, etc.).

NOTE: The list of NED functions can be found in Appendix D. The user can also extend NED with new functions.

3.6.3 Parameter References

Expressions may refer to parameters of the compound module being defined, parameters of the current module, and to parameters of already defined submodules, with the syntax `submodule.parametername` (or `submodule[index].parametername`).

Unqualified parameter names refer to a parameter of the compound module, wherever it occurs within the compound module definition. For example, all `foo` references in the following example refer to the network's `foo` parameter.

```
network Network
{
    parameters:
        double foo;
        double bar = foo;
    submodules:
        node[10]: Node {
            baz = foo;
        }
    ...
}
```

Use the **`this`** qualifier to refer to another parameter of the same submodule:

```
    submodules:
        node: Node {
            datarate = this.amount / this.duration;
    }
```

From OMNEST 5.7 onwards, there is also a **`parent`** qualifier with the obvious meaning.

NOTE: The interpretation of names which are not qualified with either **`this`** or **`parent`** and occur within submodule/channel blocks is going to change in OMNEST 6.0: An unqualified name `foo` is going to refer to the parameter of the submodule itself, i.e. will be interpreted as `this.foo`. To create NED files which are compatible with both versions, make those parameter references explicit by using the **`parent`** qualifier: `parent.foo`. A similar rule applies to the arguments of **`sizeof`** and **`exists`**.

3.6.4 Volatile Parameters

Volatile parameters are those marked with the **`volatile`** modifier keyword. Normally, expressions assigned to parameters are evaluated once, and the resulting values are stored in the parameters. In contrast, a volatile parameter holds the expression itself, and it is evaluated every time the parameter is read. Therefore, if the expression contains a stochastic or changing component, such as `normal(0,1)` (a random value from the unit normal distribution) or `simTime()` (the current simulation time), reading the parameter may yield a different value every time.

NOTE: Technically, non-volatile parameters may also contain stochastic values. However, the result of that would be that the simulation use a constant value throughout, chosen randomly at the beginning of the simulation. This is akin to running a randomly selected simulation rather than performing a Monte-Carlo simulation, hence, it is rarely desirable.

If a parameter is marked **volatile**, the C++ code that implements the corresponding module is expected to re-read the parameter every time a new value is needed, as opposed to reading it once and caching the value in a variable.

To demonstrate the use of **volatile**, suppose we have a `Queue` simple module that has a `volatile double` parameter named `serviceTime`.

```
simple Queue
{
    parameters:
        volatile double serviceTime;
}
```

Because of the **volatile** modifier, the C++ code underlying the queue module is supposed read the `serviceTime` parameter for every job serviced. Thus, if a stochastic value like `uniform(0.5s, 1.5s)` is assigned the parameter, the expression will be evaluated every time, and every job will likely have a different, random service time.

As another example, here's how one can have a time-varying parameter by exploiting the `simTime()` NED function:

```
**.serviceTime = simTime()<1000s ? 1s : 2s  # queue that slows down after 1000s
```

3.6.5 Mutable Parameters

A parameter is marked as mutable by adding the `@mutable` property to it. Mutable parameters can be set to a different value during runtime, whereas normal, i.e. non-mutable parameters cannot be changed after their initial assignment (attempts to do so will result in an error being raised).

Parameter mutability addresses the fact that although it would be technically possible to allow changing the value of any parameter to a different value during runtime, it only really makes sense to do so if the change actually takes effect. Otherwise, users doing the change could be mislead.

For example, if a module is implemented in C++ in a way that it only reads a parameter once and then uses the cached value throughout, it would be misleading to allow changing the parameter's value during simulation. For a parameter to rightfully be marked as `@mutable`, module's implementation has to be explicitly prepared to handle runtime parameter changes (see section 4.5.5).

As a practical example, a drop-tail queue module could have a `maxLength` parameter which controls the maximum number of elements the queue can hold. If it was allowed to set the `maxLength` parameter to a different value at runtime but the module would continue to operate according to the initially configured value throughout the entire simulation, that could falsify simulation results.

```
simple Queue
{
```

```
    parameters:
        int maxLength @mutable; // @mutable indicates that Queue's
                                // implementation is prepared for handling
                                // runtime changes in the value of the
                                // maximum queue length.
        ...
}
```

In a model framework that contains a large number of modules with many parameters, the presence or absence of `@mutable` allows the user to know which are the parameters whose runtime changes are properly handled by their modules. This is an important input for determining what kinds of experiments can be done with the model.

HINT: Note that although **`volatile`** and `@mutable` are two different things, parameters marked **`volatile`** may often be marked `@mutable` as well.

NOTE: `@mutable` affects backward compatibility. As it was introduced in OMNEST version 6.0, models written before that do not contain `@mutable` annotations. Such simulations models, if they rely on runtime parameter changes, may be run under OMNEST 6.0 by setting the **`parameter-mutability-check`** configuration option to `false`.

3.6.6 Units

One can declare a parameter to have an associated unit of measurement by adding the `@unit` property. An example:

```
simple App
{
    parameters:
        volatile double sendInterval @unit(s) = default(exponential(350ms));
        volatile int packetLength @unit(byte) = default(4KiB);
    ...
}
```

The `@unit(s)` and `@unit(byte)` declarations specify the measurement unit for the parameter. Values assigned to parameters must have the same or compatible unit, i.e. `@unit(s)` accepts milliseconds, nanoseconds, minutes, hours, etc., and `@unit(byte)` accepts kilobytes, megabytes, etc. as well.

NOTE: The list of units accepted by OMNEST is listed in the Appendix, see A.5.11. Unknown units (`bogomips`, etc.) can also be used, but there are no conversions for them, i.e. decimal prefixes will not be recognized.

The OMNEST runtime does a full and rigorous unit check on parameters to ensure "unit safety" of models. Constants should always include the measurement unit.

The `@unit` property of a parameter cannot be added or overridden in subclasses or in submodule declarations.

3.6.7 XML Parameters

OMNEST supports two explicit ways of passing structured data to a module using parameters: XML parameters, and object parameters with JSON-style structured data. This section describes the former, and the next one the latter.

XML parameters are declared with the keyword **xml**. When using XML parameters, OMNEST will read the XML document for you, DTD-validates it (if it contains a DOCTYPE), and presents the contents in a DOM-like object tree. It is also possible to assign a part (i.e. subtree) of the document to the parameter; the subset can be selected using an XPath-subset notation. OMNEST caches the content of the document, so it is loaded only once even if it is referenced by multiple parameters.

Values for an XML parameter can be produced using the **xmldoc()** and the **xml()** functions. **xmldoc()** accepts a filename as argument, while **xml()** parses its string argument as XML content. Of course, one can assign **xml** parameters both from NED and from `omnetpp.ini`.

The following example declares an **xml** parameter, and assigns the contents of an XML file to it. The file name is understood as being relative to the working directory.

```
simple TrafGen {
    parameters:
        xml profile;
    gates:
        output out;
}

module Node {
    submodules:
        trafGen1 : TrafGen {
            profile = xmldoc("data.xml");
        }
        ...
}
```

xmldoc() also lets one select an element *within* an XML document. In case a simulation model contains numerous modules that need XML input, this feature allows the user get rid of the many small XML files by aggregating them into a single XML file. For example, the following XML file contains two profiles identified with the IDs *gen1* and *gen2*:

```
<?xml>
<root>
    <profile id="gen1">
          <param>3</param>
          <param>5</param>
    </profile>
    <profile id="gen2">
          <param>9</param>
    </profile>
</root>
```

And one can assign each profile to a corresponding submodule using an XPath-like expression:

```
module Node {
    submodules:
```

```
        trafGen1 : TrafGen {
            profile = xmldoc("all.xml", "/root/profile[@id='gen1']");
        }
        trafGen2 : TrafGen {
            profile = xmldoc("all.xml", "/root/profile[@id='gen2']");
        }
}
```

The following example shows how specify XML content using a string literal, with the **xml()** function. This is especially useful for specifying a default value.

```
simple TrafGen {
    parameters:
        xml profile = xml("<root/>"); // empty document as default
        ...
}
```

The **xml()** function, like **xmldoc()**, also supports an optional second XPath parameter for selecting a subtree.

3.6.8 Object Parameters and Structured Data

Object parameters are declared with the keyword **object**. The values of object parameters are C++ objects, which can hold arbitrary data and can be constructed in various ways in NED. Although object parameters were introduced in OMNEST only in version 6.0, they are now the preferred way of passing structured data to modules.

There are two basic constructs in NED for creating objects: the *array* and the *object* syntax. The array syntax is a pair of square brackets that encloses the list of comma-separated array elements: *[value1, value2, ...]*. The object (a.k.a. dictionary) syntax uses curly braces around key-value pairs, the separators being colon and comma: *{ key1 : value1, key2:value2, ... }*. These constructs can be composed, so an array may contain objects and further arrays as elements, and similarly, an object may contain arrays and further objects as values, and so on. This allows describing complex data structures, with a JSON-like notation.

The notation is only JSON-*like*, as the syntax rules are more relaxed than in JSON. All valid JSON is accepted, but also some more. The main difference is that in JSON, values in arrays and objects may only be constants or `null`, while OMNEST allows NED expressions as values: quantities, **nan/inf**, parameter references, functions, arithmetic operations, etc., are all accepted.

An extra relaxation and convenience compared to strict JSON is that quotation marks around object keys may be left out, as long as the key complies with the identifier syntax.

Another extension is that for objects, the desired C++ class may be specified in front of the open curly brace: *classname { key1 : value1, ... }*. The object will be created and filled in using OMNEST's reflection features. This allows internal data structures of modules to be filled out directly, so it eliminates most of the "parsing" code which is otherwise necessary. More about this feature will be written in the chapter about C++ programming (section 4.5.4).

Object parameters with JSON-style values obsolete several workarounds that were used in pre-6.0 OMNEST versions for passing structured data to modules, for example using strings to specify numeric arrays, or using text files of ad-hoc syntax as configuration or data files. JSON-style values are also more convenient that XML input.

After this introduction, let us see some examples! We begin with a list of completely made-up object parameter assignments, to show the syntax and the possibilities:

```
simple Example {
    parameters:
        object array1 = []; // empty array
        object array2 = [2, 5, 3, -1]; // array of integers
        object array3 = [ 3, 24.5mW, "Hello", false, true ]; // misc array
        object array4 = [ nan, inf, inf s, null, nullptr ]; // special values
        object object1 = {};  // empty object
        object object2 = { foo: 100, bar: "Hello" }; // object with 2 fields
        object object3 = { "foo": 100, "bar": "Hello" }; // keys with quotes

        // composition of objects and arrays
        object array5 = [ [1,2,3], [4,5,6], [7,8,9] ];
        object array6 = [ { foo: 100, bar: "Hello" }, { baz: false } ];
        object object4 = { foo : [1,2,3], bar : [4,5,6] };
        object object5 = { obj : { foo: 1, bar: 2 }, array: [1, 2, 3 ] };

        // expression, parameter references
        double x = default(1);
        object misc = [ x, 2*x, floor(3.14), uniform(0,10) ]; // [1,2,3,?]

        // default values
        object default1 = default([]); // empty array by default
        object default2 = default({}); // empty object by default
        object default3 = default([1,2,3]); // some array by default
        object default4 = default(nullptr); // null pointer by default
}
```

The following, more practical example demonstrates how one could describe an IPv4 routing table. Each route is represented as an object, and the table itself is represented as an array of routes.

```
object routes = [
    { dest: "10.0.0.0", netmask: "255.255.0.0", interf: "eth0", metric:10 },
    { dest: "10.1.0.0", netmask: "255.255.0.0", interf: "eth1", metric:20 },
    { dest: "*", interf: "eth2" },
];
```

The next example shows the use of the extended object syntax for specifying a "template" for the packets that a traffic source module should generate. Note the stochastic expression for the `byteLength` field, and that the parameter is declared as **`volatile`**. Every time the module needs to send a packet, its C++ code should read the `packetToSend` parameter, which will cause the expression to be evaluated and a new packet of random length to be created that the module can send.

```
simple TrafficSource {
    parameters:
    volatile object packetToSend = default(cPacket {
        name: "data",
        kind: 10,
        byteLength: intuniform(64,4096)
    });
```

```
    volatile double sendInterval @unit(s) = default(exponential(100ms));
}
```

Another traffic source module that supports a predetermined schedule of what to send at which points in time could have the following parameter to describe the schedule:

```
object sendSchedule = [
    { time: 1s, pk: cPacket { name: "pk1", byteLength: 64 } },
    { time: 2s, pk: cPacket { name: "pk2", byteLength: 76 } },
    { time: 3s, pk: cPacket { name: "pk3", byteLength: 32 } },
];
```

In the next example, we want to pass a trail given with its waypoints to a module. The module will get the data in an instance of a `Trail` C++ class expressly created for this purpose. This means that the module will get the trail data in a ready-to-use form just by reading parameter, without having to do any parsing or additional processing.

We use a message file (chapter 5) to define the classes; the C++ classes will be automatically generated by OMNEST from it.

```
// file: Trail.msg
struct Point {
    double x;
    double y;
}

class Trail extends cObject {
    Point waypoints[];
}
```

An actual trail can be specified in NED like this:

```
object trail = Trail {
    waypoints: [
        { x: 1, y : 5 },
        { x: 4, y : 6 },
        { x: 3, y : 8 },
        { x: 5, y : 3 }
    ]
  };
```

Values for object parameters may also be placed in ini files, just like values for other parameter types. In ini files, indented lines are treated as continuations of the previous line, so the above example doesn't need trailing backslashes when moved to `omnetpp.ini`:

```
**.trail = Trail {
        waypoints: [
            { x: 1, y : 5 },
            { x: 4, y : 6 },
            { x: 3, y : 8 },
            { x: 5, y : 3 }
        ]
      }
```

3.6.9 Passing a Formula as Parameter

The special operator **expr()** allows one to pass a formula into a module as a parameter. **expr()** takes an expression as argument, which *syntactically* must correspond to the general syntax of NED expressions. However, it is not a normal NED expression: it will *not* be interpreted and evaluated as one. Instead, it will be encapsulated into, and returned as, an object, and typically assigned to a module parameter.

The module may access the object via the parameter, and may evaluate the expression encapsulated in it any number of times during simulation. While doing so, the module's code can freely determine how various identifiers and other syntactical elements in the expression are to be interpreted.

Let us see a practical example. In the model of a wireless network, one of the tasks is to compute the path loss suffered by each wirelessly transmitted frame, as part of the procedure to determine whether the frame could be successfully received by the receiver node. There are several formulas for computing the path loss (free space, two-ray ground reflection, etc), and it depends on multiple factors which one to use. If the model author wants to leave it open for their users to specify the formula they want to use, they might define the model like so:

```
simple RadioMedium {
    parameters:
        object pathLoss; // =expr(...): formula to compute path loss
    ...
}
```

The `pathLoss` parameter expects the formula to be given with **expr()**. The formula is expected to contain two variables, `distance` and `frequency`, which stand for the distance between the transmitter and the receiver and the packet transmission frequency, respectively. The module would evaluate the expression for each frame, binding values that correspond to the current frame to those variables.

Given the above, free space path loss would be specified to the module with the following formula (assuming isotropic antennas with the same polarization, etc.):

```
**.pathLoss = expr((4 * 3.14159 * distance * frequency / c) ^ 2)
```

The next example is borrowed from the INET Framework, which extensively uses **expr()** for specifying packet filter conditions. A few examples:

```
expr(hasBitError)
expr(name == 'P1')
expr(name =~ 'P*')
expr(totalLength == 128B)
expr(ipv4.destAddress.str() == '10.0.0.1' && udp.destPort == 42)
```

The interesting part is that the packet itself does not appear explicitly in the expressions. Instead, identifiers like `hasBitError` and `name` are interpreted as attributes of the packet, as if the user had written e.g. `pk.hasBitError` and `pk.name`. Similarly, `ipv4` and `udp` stand for the IPv4 and UDP headers of the packet. The last line also shows that the interpretation of member accesses and method calls is also in the hands of the module's code.

The details of implementing **expr()** support in modules will be described as part of the simulation library, in section 7.8.

3.7 Gates

Gates are the connection points of modules. OMNEST has three types of gates: *input*, *output* and *inout*, the latter being essentially an input and an output gate glued together.

A gate, whether input or output, can only be connected to one other gate. (For compound module gates, this means one connection "outside" and one "inside".) It is possible, though generally not recommended, to connect the input and output sides of an inout gate separately (see section 3.9).

One can create single gates and gate vectors. The size of a gate vector can be given inside square brackets in the declaration, but it is also possible to leave it open by just writing a pair of empty brackets ("`[]`").

When the gate vector size is left open, one can still specify it later, when subclassing the module, or when using the module for a submodule in a compound module. However, it does not need to be specified because one can create connections with the *gate*`++` operator that automatically expands the gate vector.

The gate size can be queried from various NED expressions with the `sizeof()` operator.

NED normally requires that all gates be connected. To relax this requirement, one can annotate selected gates with the `@loose` property, which turns off the connectivity check for that gate. Also, input gates that solely exist so that the module can receive messages via `sendDirect()` (see 4.7.5) should be annotated with `@directIn`. It is also possible to turn off the connectivity check for all gates within a compound module by specifying the **`allowunconnected`** keyword in the module's connections section.

Let us see some examples.

In the following example, the `Classifier` module has one input for receiving jobs, which it will send to one of the outputs. The number of outputs is determined by a module parameter:

```
simple Classifier {
    parameters:
        int numCategories;
    gates:
        input in;
        output out[numCategories];
}
```

The following `Sink` module also has its `in[]` gate defined as a vector, so that it can be connected to several modules:

```
simple Sink {
    gates:
        input in[];
}
```

The following lines define a node for building a square grid. Gates around the edges of the grid are expected to remain unconnected, hence the `@loose` annotation:

```
simple GridNode {
    gates:
        inout neighbour[4] @loose;
}
```

`WirelessNode` below is expected to receive messages (radio transmissions) via direct sending, so its `radioIn` gate is marked with `@directIn`.

```
simple WirelessNode {
    gates:
        input radioIn @directIn;
}
```

In the following example, we define `TreeNode` as having gates to connect any number of children, then subclass it to get a `BinaryTreeNode` to set the gate size to two:

```
simple TreeNode {
    gates:
        inout parent;
        inout children[];
}

simple BinaryTreeNode extends TreeNode {
    gates:
        children[2];
}
```

An example for setting the gate vector size in a submodule, using the same `TreeNode` module type as above:

```
module BinaryTree {
    submodules:
        nodes[31]: TreeNode {
            gates:
                children[2];
        }
    connections:
        ...
}
```

3.8 Submodules

Modules that a compound module is composed of are called its submodules. A submodule has a name, and it is an instance of a compound or simple module type. In the NED definition of a submodule, this module type is usually given statically, but it is also possible to specify the type with a string expression. (The latter feature, *parametric submodule types*, will be discussed in section 3.11.1.)

NED supports submodule arrays (vectors) and conditional submodules as well. Submodule vector size, unlike gate vector size, must always be specified and cannot be left open as with gates.

It is possible to add new submodules to an existing compound module via subclassing; this has been described in the section 3.4.

The basic syntax of submodules is shown below:

```
module Node
{
```

```
    submodules:
        routing: Routing;   // a submodule
        queue[sizeof(port)]: Queue;  // submodule vector
        ...
}
```

As already seen in previous code examples, a submodule may also have a curly brace block as body, where one can assign parameters, set the size of gate vectors, and add/modify properties like the display string (`@display`). It is not possible to add new parameters and gates.

Display strings specified here will be merged with the display string from the type to get the effective display string. The merge algorithm is described in chapter 8.

```
module Node
{
    gates:
        inout port[];
    submodules:
        routing: Routing {
            parameters:   // this keyword is optional
                routingTable = "routingtable.txt"; // assign parameter
            gates:
                in[sizeof(port)];  // set gate vector size
                out[sizeof(port)];
        }
        queue[sizeof(port)]: Queue {
            @display("t=queue id $id"); // modify display string
            id = 1000+index;  // use submodule index to generate different IDs
        }
    connections:
        ...
}
```

An empty body may be omitted, that is,

```
      queue: Queue;
```

is the same as

```
      queue: Queue {
      }
```

A submodule or submodule vector can be conditional. The **`if`** keyword and the condition itself goes after the submodule type, like in the example below:

```
module Host
{
    parameters:
        bool withTCP = default(true);
    submodules:
        tcp : TCP if withTCP;
        ...
}
```

Note that with submodule vectors, setting zero vector size can be used as an alternative to the **`if`** condition.

3.9 Connections

Connections are defined in the **connections** section of compound modules. Connections cannot span across hierarchy levels; one can connect two submodule gates, a submodule gate and the "inside" of the parent (compound) module's gates, or two gates of the parent module (though this is rarely useful), but it is not possible to connect to any gate outside the parent module, or inside compound submodules.

Input and output gates are connected with a normal arrow, and inout gates with a double-headed arrow "<-->". To connect the two gates with a channel, use two arrows and put the channel specification in between. The same syntax is used to add properties such as `@display` to the connection.

Some examples have already been shown in the NED Quickstart section (3.2); let's see some more.

It has been mentioned that an inout gate is basically an input and an output gate glued together. These sub-gates can also be addressed (and connected) individually if needed, as `port$i` and `port$o` (or for vector gates, as `port$i[`*k*`]` and `port$o[`*k*`]`).

Gates are specified as *modulespec.gatespec* (to connect a submodule), or as *gatespec* (to connect the compound module). *modulespec* is either a submodule name (for scalar submodules), or a submodule name plus an index in square brackets (for submodule vectors). For scalar gates, *gatespec* is the gate name; for gate vectors it is either the gate name plus an index in square brackets, or *gatename*`++`.

The *gatename*`++` notation causes the first unconnected gate index to be used. If all gates of the given gate vector are connected, the behavior is different for submodules and for the enclosing compound module. For submodules, the gate vector expands by one. For a compound module, after the last gate is connected, `++` will stop with an error.

> **NOTE:** Why is it not possible to expand a gate vector of the compound module? The model structure is built in top-down order, so new gates would be left unconnected on the outside, as there is no way in NED to "go back" and connect them afterwards.

When the `++` operator is used with `$i` or `$o` (e.g. `g$i++` or `g$o++`, see later), it will actually add a gate pair (input+output) to maintain equal gate sizes for the two directions.

3.9.1 Channel Specification

Channel specifications (`-->`*channelspec*`-->` inside a connection) are similar to submodules in many respect. Let's see some examples!

The following connections use two user-defined channel types, `Ethernet100` and `Backbone`. The code shows the syntax for assigning parameters (`cost` and `length`) and specifying a display string (and NED properties in general):

```
a.g++ <--> Ethernet100 <--> b.g++;
a.g++ <--> Backbone {cost=100; length=52km; ber=1e-8;} <--> b.g++;
a.g++ <--> Backbone {@display("ls=green,2");} <--> b.g++;
```

When using built-in channel types, the type name can be omitted; it will be inferred from the parameter names.

```
a.g++ <--> {delay=10ms;} <--> b.g++;
```

```
a.g++ <--> {delay=10ms; ber=1e-8;} <--> b.g++;
a.g++ <--> {@display("ls=red");} <--> b.g++;
```

If `datarate`, `ber` or `per` is assigned, `ned.DatarateChannel` will be chosen. Otherwise, if `delay` or `disabled` is present, it will be `ned.DelayChannel`; otherwise it is `ned.IdealChannel`. Naturally, if other parameter names are assigned in a connection without an explicit channel type, it will be an error (with *"ned.DelayChannel has no such parameter"* or similar message).

Connection parameters, similarly to submodule parameters, can also be assigned using pattern assignments, albeit the channel names to be matched with patterns are a little more complicated and less convenient to use. A channel can be identified with the name of its source gate plus the channel name; the channel name is currently always `channel`. It is illustrated by the following example:

```
module Queueing
{
    parameters:
        source.out.channel.delay = 10ms;
        queue.out.channel.delay = 20ms;
    submodules:
        source: Source;
        queue: Queue;
        sink: Sink;
    connections:
        source.out --> ned.DelayChannel --> queue.in;
        queue.out --> ned.DelayChannel <--> sink.in;
```

Using bidirectional connections is a bit trickier, because both directions must be covered separately:

```
network Network
{
    parameters:
        hostA.g$o[0].channel.datarate = 100Mbps; // the A -> B connection
        hostB.g$o[0].channel.datarate = 100Mbps; // the B -> A connection
        hostA.g$o[1].channel.datarate = 1Gbps;   // the A -> C connection
        hostC.g$o[0].channel.datarate = 1Gbps;   // the C -> A connection
    submodules:
        hostA: Host;
        hostB: Host;
        hostC: Host;
    connections:
        hostA.g++ <--> ned.DatarateChannel <--> hostB.g++;
        hostA.g++ <--> ned.DatarateChannel <--> hostC.g++;
```

Also, with the `++` syntax it is not always easy to figure out which gate indices map to the connections one needs to configure. If connection objects could be given names to override the default name "`channel`", that would make it easier to identify connections in patterns. This feature is described in the next section.

3.9.2 Channel Names

The default name given to channel objects is `"channel"`. Since OMNEST 4.3 it is possible to specify the name explicitly, and also to override the default name per channel type. The

purpose of custom channel names is to make addressing easier when channel parameters are assigned from ini files.

The syntax for naming a channel in a connection is similar to submodule syntax: *name: type*. Since both *name* and *type* are optional, the colon must be there after *name* even if *type* is missing, in order to remove the ambiguity.

Examples:

```
r1.pppg++ <--> eth1: EthernetChannel <--> r2.pppg++;
a.out --> foo: {delay=1ms;} --> b.in;
a.out --> bar: --> b.in;
```

In the absence of an explicit name, the channel name comes from the `@defaultname` property of the channel type if that exists.

```
channel Eth10G extends ned.DatarateChannel like IEth {
    @defaultname(eth10G);
}
```

There's a catch with `@defaultname` though: if the channel type is specified with a `**.`*channelname*`.liketype=` line in an ini file, then the channel type's `@defaultname` cannot be used as *channelname* in that configuration line, because the channel type would only be known as a result of using that very configuration line. To illustrate the problem, consider the above `Eth10G` channel, and a compound module containing the following connection:

```
r1.pppg++ <--> <> like IEth <--> r2.pppg++;
```

Then consider the following inifile:

```
**.eth10G.typename = "Eth10G"    # Won't match! The eth10G name would come from
                                 #   the Eth10G type - catch-22!
**.channel.typename = "Eth10G"   # OK, as lookup assumes the name "channel"
**.eth10G.datarate = 10.01Gbps   # OK, channel already exists with name "eth10G"
```

The anomaly can be avoided by using an explicit channel name in the connection, not using `@defaultname`, or by specifying the type via a module parameter (e.g. writing `<param> like ...` instead of `<> like ...`).

3.10 Multiple Connections

Simple programming constructs (loop, conditional) allow creating multiple connections easily.

This will be shown in the following examples.

3.10.1 Examples

Chain

One can create a chain of modules like this:

```
module Chain
    parameters:
        int count;
```

```
    submodules:
        node[count] : Node {
            gates:
                port[2];
        }
    connections allowunconnected:
        for i = 0..count-2 {
            node[i].port[1] <--> node[i+1].port[0];
        }
}
```

Binary Tree

One can build a binary tree in the following way:

```
simple BinaryTreeNode {
    gates:
        inout left;
        inout right;
        inout parent;
}

module BinaryTree {
    parameters:
        int height;
    submodules:
        node[2^height-1]: BinaryTreeNode;
    connections allowunconnected:
        for i=0..2^(height-1)-2 {
            node[i].left <--> node[2*i+1].parent;
            node[i].right <--> node[2*i+2].parent;
        }
}
```

Note that not every gate of the modules will be connected. By default, an unconnected gate produces a run-time error message when the simulation is started, but this error message is turned off here with the **allowunconnected** modifier. Consequently, it is the simple modules' responsibility not to send on an unconnected gate.

Random Graph

Conditional connections can be used to generate random topologies, for example. The following code generates a random subgraph of a full graph:

```
module RandomGraph {
    parameters:
        int count;
        double connectedness; // 0.0<x<1.0
    submodules:
        node[count]: Node {
            gates:
```

```
                in[count];
                out[count];
        }
    connections allowunconnected:
        for i=0..count-1, for j=0..count-1 {
            node[i].out[j] --> node[j].in[i]
                if i!=j && uniform(0,1)<connectedness;
        }
}
```

Note the use of the **allowunconnected** modifier here too, to turn off error messages produced by the network setup code for unconnected gates.

3.10.2 Connection Patterns

Several approaches can be used for creating complex topologies that have a regular structure; three of them are described below.

"Subgraph of a Full Graph"

This pattern takes a subset of the connections of a full graph. A condition is used to "carve out" the necessary interconnection from the full graph:

```
for i=0..N-1, for j=0..N-1 {
    node[i].out[...] --> node[j].in[...] if condition(i,j);
}
```

The RandomGraph compound module (presented earlier) is an example of this pattern, but the pattern can generate any graph where an appropriate $condition(i,j)$ can be formulated. For example, when generating a tree structure, the condition would return whether node j is a child of node i or vice versa.

Though this pattern is very general, its usage can be prohibitive if the number of nodes N is high and the graph is sparse (it has much less than N^2 connections). The following two patterns do not suffer from this drawback.

"Connections of Each Node"

The pattern loops through all nodes and creates the necessary connections for each one. It can be generalized like this:

```
for i=0..Nnodes, for j=0..Nconns(i)-1 {
    node[i].out[j] --> node[rightNodeIndex(i,j)].in[j];
}
```

The Hypercube compound module (to be presented later) is a clear example of this approach. BinaryTree can also be regarded as an example of this pattern where the inner j loop is unrolled.

The applicability of this pattern depends on how easily the $rightNodeIndex(i,j)$ function can be formulated.

"Enumerate All Connections"

A third pattern is to list all connections within a loop:

```
for i=0..Nconnections-1 {
    node[leftNodeIndex(i)].out[...] --> node[rightNodeIndex(i)].in[...];
}
```

This pattern can be used if $leftNodeIndex(i)$ and $rightNodeIndex(i)$ mapping functions can be sufficiently formulated.

The `Chain` module is an example of this approach where the mapping functions are extremely simple: $leftNodeIndex(i) = i$ and $rightNodeIndex(i) = i + 1$. The pattern can also be used to create a random subset of a full graph with a fixed number of connections.

In the case of irregular structures where none of the above patterns can be employed, one can resort to listing all connections, like one would do it in most existing simulators.

3.11 Parametric Submodule and Connection Types

3.11.1 Parametric Submodule Types

A submodule type may be specified with a module parameter of the type **`string`**, or in general, with any string-typed expression. The syntax uses the **`like`** keyword.

Let us begin with an example:

```
network Net6
{
    parameters:
        string nodeType;
    submodules:
        node[6]: <nodeType> like INode {
            address = index;
        }
    connections:
        ...
}
```

It creates a submodule vector whose module type will come from the `nodeType` parameter. For example, if `nodeType` is set to `"SensorNode"`, then the module vector will consist of sensor nodes, provided such module type exists and it qualifies. What this means is that the `INode` must be an existing *module interface*, which the `SensorNode` module type must implement (more about this later).

As already mentioned, one can write an expression between the angle brackets. The expression may use the parameters of the parent module and of previously defined submodules, and has to yield a string value. For example, the following code is also valid:

```
network Net6
{
    parameters:
        string nodeTypePrefix;
        int variant;
```

```
    submodules:
        node[6]: <nodeTypePrefix + "Node" + string(variant)> like INode {
            ...
}
```

The corresponding NED declarations:

```
moduleinterface INode
{
    parameters:
        int address;
    gates:
        inout port[];
}

module SensorNode like INode
{
    parameters:
        int address;
        ...
    gates:
        inout port[];
        ...
}
```

The "`<nodeType> like INode`" syntax has an issue when used with submodule vectors: does not allow one to specify different types for different indices. The following syntax is better suited for submodule vectors:

The expression between the angle brackets may be left out altogether, leaving a pair of empty angle brackets, `<>`:

```
module Node
{
    submodules:
        nic: <> like INic;  // type name expression left unspecified
        ...
}
```

Now the submodule type name is expected to be defined via typename pattern assignments. Typename pattern assignments look like pattern assignments for the submodule's parameters, only the parameter name is replaced by the **`typename`** keyword. Typename pattern assignments may also be written in the configuration file. In a network that uses the above `Node` NED type, typename pattern assignments would look like this:

```
network Network
{
    parameters:
        node[*].nic.typename = "Ieee80211g";
    submodules:
        node: Node[100];
}
```

A default value may also be specified between the angle brackets; it will be used if there is no typename assignment for the module:

```
module Node
{
    submodules:
        nic: <default("Ieee80211b")> like INic;
        ...
}
```

There must be exactly one module type that goes by the simple name `Ieee80211b` and also implements the module interface `INic`, otherwise an error message will be issued. (The imports in `Node`'s the NED file play no role in the type resolution.) If there are two or more such types, one can remove the ambiguity by specifying the fully qualified module type name, i.e. one that also includes the package name:

```
module Node
{
    submodules:
        nic: <default("acme.wireless.Ieee80211b")> like INic; // made-up name
        ...
}
```

3.11.2 Conditional Parametric Submodules

When creating reusable compound modules, it is often useful to be able to make a parametric submodule also optional. One solution is to let the user define the submodule type with a string parameter, and not create the module when the parameter is set to the empty string. Like this:

```
module Node
{
    parameters:
        string tcpType = default("Tcp");
    submodules:
        tcp: <tcpType> like ITcp if tcpType!="";
}
```

However, this pattern, when used extensively, can lead to a large number of string parameters. Luckily, it is also possible to achieve the same effect with **typename**, without using extra parameters:

```
module Node
{
    submodules:
        tcp: <default("Tcp")> like ITcp if typename!="";
}
```

The **typename** operator in a submodule's **if** condition evaluates to the would-be type of the submodule. By using the `typename!=""` condition, we can let the user eliminate the `tcp` submodule by setting its typename to the empty string. For example, in a network that uses the above NED type, typename pattern assignments could look like this:

```
network Network
{
    parameters:
```

```
        node1.tcp.typename = "TcpExt"; // let node1 use a custom TCP
        node2.tcp.typename = ""; // no TCP in node2
    submodules:
        node1: Node;
        node2: Node;
}
```

Note that this trick does not work with submodule vectors. The reason is that the condition applies to the vector as a whole, while type is per-element.

It is often also useful to be able to check, e.g. in the connections section, whether a conditional submodule has been created or not. This can be done with the **`exists()`** operator. An example:

```
module Node
{
        ...
    connections:
        ip.tcpOut --> tcp.ipIn if exists(ip) && exists(tcp);
}
```

Limitation: **`exists()`** may only be used *after* the submodule's occurrence in the compound module.

3.11.3 Parametric Connection Types

Parametric connection types work similarly to parametric submodule types, and the syntax is similar as well. A basic example that uses a parameter of the parent module:

```
a.g++ <--> <channelType> like IMyChannel <--> b.g++;
a.g++ <--> <channelType> like IMyChannel {@display("ls=red");} <--> b.g++;
```

The expression may use loop variables, parameters of the parent module and also parameters of submodules (e.g. `host[2].channelType`).

The type expression may also be absent, and then the type is expected to be specified using typename pattern assignments:

```
a.g++ <--> <> like IMyChannel <--> b.g++;
a.g++ <--> <> like IMyChannel {@display("ls=red");} <--> b.g++;
```

A default value may also be given:

```
a.g++ <--> <default("Ethernet100")> like IMyChannel <--> b.g++;
a.g++ <--> <default(channelType)> like IMyChannel <--> b.g++;
```

The corresponding type pattern assignments:

```
a.g$o[0].channel.typename = "Ethernet1000";  // A -> B channel
b.g$o[0].channel.typename = "Ethernet1000";  // B -> A channel
```

3.12 Metadata Annotations (Properties)

NED properties are metadata annotations that can be added to modules, parameters, gates, connections, NED files, packages, and virtually anything in NED. `@display`, `@class`, `@names-`

pace, @mutable, @unit, @prompt, @loose, @directIn are all properties that have been mentioned in previous sections, but those examples only scratch the surface of what properties are used for.

Using properties, one can attach extra information to NED elements. Some properties are interpreted by NED, by the simulation kernel; other properties may be read and used from within the simulation model, or provide hints for NED editing tools.

Properties are attached to the type, so one cannot have different properties defined per-instance. All instances of modules, connections, parameters, etc. created from any particular location in the NED files have identical properties.

The following example shows the syntax for annotating various NED elements:

```
@namespace(foo);  // file property

module Example
{
    parameters:
       @node;   // module property
       @display("i=device/pc");   // module property
       int a @unit(s) = default(1); // parameter property
    gates:
       output out @loose @labels(pk);  // gate properties
    submodules:
       src: Source {
           parameters:
              @display("p=150,100");  // submodule property
              count @prompt("Enter count:"); // adding a property to a parameter
           gates:
              out[] @loose;  // adding a property to a gate
       }
       ...
    connections:
       src.out++ --> { @display("ls=green,2"); } --> sink1.in; // connection prop.
       src.out++ --> Channel { @display("ls=green,2"); } --> sink2.in;
}
```

3.12.1 Property Indices

Sometimes it is useful to have multiple properties with the same name, for example for declaring multiple statistics produced by a simple module. *Property indices* make this possible.

A property index is an identifier or a number in square brackets after the property name, such as eed and jitter in the following example:

```
simple App {
    @statistic[eed](title="end-to-end delay of received packets";unit=s);
    @statistic[jitter](title="jitter of received packets");
}
```

This example declares two statistics as @statistic properties, @statistic[eed] and @statistic[jitter]. Property values within the parentheses are used to supply additional info, like a more descriptive name (title="..." or a unit (unit=s). Property indices can be conve-

niently accessed from the C++ API as well; for example it is possible to ask what indices exist for the "statistic" property, and it will return a list containing "eed" and "jitter").

In the @statistic example the index was textual and meaningful, but neither is actually required. The following dummy example shows the use of numeric indices which may be ignored altogether by the code that interprets the properties:

```
simple Dummy {
    @foo[1](what="apples";amount=2);
    @foo[2](what="oranges";amount=5);
}
```

Note that without the index, the lines would actually define the same @foo property, and would overwrite each other's values.

Indices also make it possible to override entries via inheritance:

```
simple DummyExt extends Dummy {
    @foo[2](what="grapefruits"); // 5 grapefruits instead of 5 oranges
}
```

3.12.2 Data Model

Properties may contain data, given in parentheses; the data model is quite flexible. To begin with, properties may contain no value or a single value:

```
@node;
@node(); // same as @node
@class(FtpApp2);
```

Properties may contain lists:

```
@foo(Sneezy,Sleepy,Dopey,Doc,Happy,Bashful,Grumpy);
```

They may contain key-value pairs, separated by semicolons:

```
@foo(x=10.31; y=30.2; unit=km);
```

In key-value pairs, each value can be a (comma-separated) list:

```
@foo(coords=47.549,19.034;labels=vehicle,router,critical);
```

The above examples are special cases of the general data model. According to the data model, properties contain *key-valuelist* pairs, separated by semicolons. Items in *valuelist* are separated by commas. Wherever *key* is missing, values go on the valuelist of the *default key*, the empty string.

Value items may contain words, numbers, string constants and some other characters, but not arbitrary strings. Whenever the syntax does not permit some value, it should be enclosed in quotes. This quoting does not affect the value because the parser automatically drops one layer of quotes; thus, @class(TCP) and @class("TCP") are exactly the same. If the quotes themselves need to be part of the value, an extra layer of quotes and escaping are the solution: @foo("\"some string\"").

There are also some conventions. One can use properties to tag NED elements; for example, a @host property could be used to mark all module types that represent various hosts. This

property could be recognized e.g. by editing tools, by topology discovery code inside the simulation model, etc.

The convention for such a "marker" property is that any extra data in it (i.e. within parens) is ignored, except a single word `false`, which has the special meaning of "turning off" the property. Thus, any simulation model or tool that interprets properties should handle all the following forms as equivalent to `@host`: `@host()`, `@host(true)`, `@host(anything-but-false)`, `@host(a=1;b=2)`; and `@host(false)` should be interpreted as the lack of the `@host` tag.

3.12.3 Overriding and Extending Property Values

Properties defined on a module or channel type may be updated both by subclassing and when using type as a submodule or connection channel. One can add new properties, and also modify existing ones.

When modifying a property, the new property is merged with the old one. The rules of merging are fairly simple. New keys simply get added. If a key already exists in the old property, items in its valuelist overwrite items on the same position in the old property. A single hyphen (–) as valuelist item serves as "antivalue", it removes the item at the corresponding position.

Some examples:

base	`@prop`
new	`@prop(a)`
result	`@prop(a)`

base	`@prop(a,b,c)`
new	`@prop(,-)`
result	`@prop(a,,c)`

base	`@prop(foo=a,b)`
new	`@prop(foo=A,,c;bar=1,2)`
result	`@prop(foo=A,b,c;bar=1,2)`

NOTE: The above merge rules are part of NED, but the code that interprets properties may have special rules for certain properties. For example, the `@unit` property of parameters is not allowed to be overridden, and `@display` is merged with special although similar rules (see Chapter 8).

3.13 Inheritance

Inheritance support in the NED language is only described briefly here, because several details and examples have been already presented in previous sections.

In NED, a type may only extend (**`extends`** keyword) an element of the same component type: a simple module may extend a simple module, a channel may extend a channel, a module interface may extend a module interface, and so on. There is one irregularity, however: A compound module may extend a simple module (and inherits its C++ class), but not vica versa.

Single inheritance is supported for modules and channels, and multiple inheritance is supported for module interfaces and channel interfaces. A network is a shorthand for a compound

module with the `@isNetwork` property set, so the same rules apply to it as to compound modules.

However, a simple or compound module type may implement (**`like`** keyword) several module interfaces; likewise, a channel type may implement several channel interfaces.

> **IMPORTANT:** When you extend a simple module type both in NED and in C++, you must use the `@class` property to tell NED to use the new C++ class – otherwise the new module type inherits the C++ class of the base!

Inheritance may:

- add new properties, parameters, gates, inner types, submodules, connections, as long as names do not conflict with inherited names
- modify inherited properties, and properties of inherited parameters and gates
- it may not modify inherited submodules, connections and inner types

For details and examples, see the corresponding sections of this chapter (simple modules 3.3, compound modules 3.4, channels 3.5, parameters 3.6, gates 3.7, submodules 3.8, connections 3.9, module interfaces and channel interfaces 3.11.1).

3.14 Packages

Having all NED files in a single directory is fine for small simulation projects. When a project grows, however, it sooner or later becomes necessary to introduce a directory structure, and sort the NED files into them. NED natively supports directory trees with NED files, and calls directories *packages*. Packages are also useful for reducing name conflicts, because names can be qualified with the package name.

> **NOTE:** NED packages are based on the Java package concept, with minor enhancements. If you are familiar with Java, you'll find little surprise in this section.

3.14.1 Overview

When a simulation is run, one must tell the simulation kernel the directory which is the root of the package tree; let's call it *NED source folder*. The simulation kernel will traverse the whole directory tree, and load all NED files from every directory. One can have several NED directory trees, and their roots (the NED source folders) should be given to the simulation kernel in the *NED path* variable. The NED path can be specified in several ways: as an environment variable (`NEDPATH`), as a configuration option (**`ned-path`**), or as a command-line option to the simulation runtime (`-n`). `NEDPATH` is described in detail in chapter 11.

Directories in a NED source tree correspond to packages. If NED files are in the `<root>/a/b/c` directory (where `<root>` is listed in NED path), then the package name is `a.b.c`. The package name has to be explicitly declared at the top of the NED files as well, like this:

```
package a.b.c;
```

The package name that follows from the directory name and the declared package must match; it is an error if they don't. (The only exception is the root `package.ned` file, as described below.)

By convention, package names are all lowercase, and begin with either the project name (`myproject`), or the reversed domain name plus the project name (`org.example.myproject`). The latter convention would cause the directory tree to begin with a few levels of empty directories, but this can be eliminated with a toplevel `package.ned`.

NED files called `package.ned` have a special role, as they are meant to represent the whole package. For example, comments in `package.ned` are treated as documentation of the package. Also, a `@namespace` property in a `package.ned` file affects all NED files in that directory and all directories below.

The toplevel `package.ned` file can be used to designate the root package, which is useful for eliminating a few levels of empty directories resulting from the package naming convention. For example, given a project where all NED types are under the `org.acme.foosim` package, one can eliminate the empty directory levels `org`, `acme` and `foosim` by creating a `package.ned` file in the source root directory with the package declaration `org.example.myproject`. This will cause a directory `foo` under the root to be interpreted as package `org.example.myproject.foo`, and NED files in them must contain that as package declaration. Only the root `package.ned` can define the package, `package.ned` files in subdirectories must follow it.

Let's look at the INET Framework as example, which contains hundreds of NED files in several dozen packages. The directory structure looks like this:

```
INET/
    src/
        base/
        transport/
            tcp/
            udp/
            ...
        networklayer/
        linklayer/
        ...
    examples/
        adhoc/
        ethernet/
        ...
```

The `src` and `examples` subdirectories are denoted as NED source folders, so `NEDPATH` is the following (provided INET was unpacked in `/home/joe`):

```
/home/joe/INET/src;/home/joe/INET/examples
```

Both `src` and `examples` contain `package.ned` files to define the root package:

```
// INET/src/package.ned:
package inet;
```

```
// INET/examples/package.ned:
package inet.examples;
```

And other NED files follow the package defined in `package.ned`:

```
// INET/src/transport/tcp/TCP.ned:
package inet.transport.tcp;
```

3.14.2 Name Resolution, Imports

We already mentioned that packages can be used to distinguish similarly named NED types. The name that includes the package name (`a.b.c.Queue` for a `Queue` module in the `a.b.c` package) is called *fully qualified name*; without the package name (`Queue`) it is called *simple name*.

Simple names alone are not enough to unambiguously identify a type. Here is how one can refer to an existing type:

1. By fully qualified name. This is often cumbersome though, as names tend to be too long;
2. Import the type, then the simple name will be enough;
3. If the type is in the same package, then it doesn't need to be imported; it can be referred to by simple name

Types can be imported with the **`import`** keyword by either fully qualified name, or by a wildcard pattern. In wildcard patterns, one asterisk ("`*`") stands for "any character sequence not containing period", and two asterisks ("`**`") mean "any character sequence which may contain period".

So, any of the following lines can be used to import a type called `inet.protocols.networklayer.ip.RoutingTable`:

```
import inet.protocols.networklayer.ip.RoutingTable;
import inet.protocols.networklayer.ip.*;
import inet.protocols.networklayer.ip.Ro*Ta*;
import inet.protocols.*.ip.*;
import inet.**.RoutingTable;
```

If an import explicitly names a type with its exact fully qualified name, then that type must exist, otherwise it is an error. Imports containing wildcards are more permissive, it is allowed for them not to match any existing NED type (although that might generate a warning.)

Inner types may not be referred to outside their enclosing types, so they cannot be imported either.

3.14.3 Name Resolution With "like"

The situation is a little different for submodule and connection channel specifications using the **`like`** keyword, when the type name comes from a string-valued expression (see section 3.11.1 about submodule and channel types as parameters). Imports are not much use here: at the time of writing the NED file it is not yet known what NED types will be suitable for being "plugged in" there, so they cannot be imported in advance.

There is no problem with fully qualified names, but simple names need to be resolved differently. What NED does is this: it determines which interface the module or channel type must implement (i.e. `... like INode`), and then collects the types that have the given simple name AND implement the given interface. There must be exactly one such type, which is then used. If there is none or there are more than one, it will be reported as an error.

Let us see the following example:

```
module MobileHost
{
    parameters:
        string mobilityType;
    submodules:
        mobility: <mobilityType> like IMobility;
        ...
}
```

and suppose that the following modules implement the `IMobility` module interface: `inet.mobility.RandomWalk`, `inet.adhoc.RandomWalk`, `inet.mobility.MassMobility`. Also suppose that there is a type called `inet.examples.adhoc.MassMobility` but it does not implement the interface.

So if `mobilityType="MassMobility"`, then `inet.mobility.MassMobility` will be selected; the other `MassMobility` doesn't interfere. However, if `mobilityType="RandomWalk"`, then it is an error because there are two matching `RandomWalk` types. Both `RandomWalk`'s can still be used, but one must explicitly choose one of them by providing a package name: `mobilityType="inet.adhoc.RandomWalk"`.

3.14.4 The Default Package

It is not mandatory to make use of packages: if all NED files are in a single directory listed on the NEDPATH, then package declarations (and imports) can be omitted. Those files are said to be in the *default package*.

Chapter 4

Simple Modules

Simple modules are the active components in the model. Simple modules are programmed in C++, using the OMNEST class library. The following sections contain a short introduction to discrete event simulation in general, explain how its concepts are implemented in OMNEST, and give an overview and practical advice on how to design and code simple modules.

4.1 Simulation Concepts

This section contains a very brief introduction into how discrete event simulation (DES) works, in order to introduce terms we'll use when explaining OMNEST concepts and implementation.

4.1.1 Discrete Event Simulation

A *discrete event system* is a system where state changes (events) happen at discrete instances in time, and events take zero time to happen. It is assumed that nothing (i.e. nothing interesting) happens between two consecutive events, that is, no state change takes place in the system between the events. This is in contrast to *continuous* systems where state changes are continuous. Systems that can be viewed as discrete event systems can be modeled using discrete event simulation, DES.

For example, computer networks are usually viewed as discrete event systems. Some of the events are:

- start of a packet transmission
- end of a packet transmission
- expiry of a retransmission timeout

This implies that between two events such as *start of a packet transmission* and *end of a packet transmission*, nothing interesting happens. That is, the packet's state remains *being transmitted*. Note that the definition of "interesting" events and states always depends on the intent and purposes of the modeler. If we were interested in the transmission of individual bits, we would have included something like *start of bit transmission* and *end of bit transmission* among our events.

The time when events occur is often called *event timestamp*; with OMNEST we use the term *arrival time* (because in the class library, the word "timestamp" is reserved for a user-settable attribute in the event class). Time within the model is often called *simulation time*, *model time* or *virtual time* as opposed to real time or CPU time which refer to how long the simulation program has been running and how much CPU time it has consumed.

4.1.2 The Event Loop

Discrete event simulation maintains the set of future events in a data structure often called FES (Future Event Set) or FEL (Future Event List). Such simulators usually work according to the following pseudocode:

```
initialize -- this includes building the model and
              inserting initial events to FES

while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    process event
    (processing may insert new events in FES or delete existing ones)
}
finish simulation (write statistical results, etc.)
```

The initialization step usually builds the data structures representing the simulation model, calls any user-defined initialization code, and inserts initial events into the FES to ensure that the simulation can start. Initialization strategies can differ considerably from one simulator to another.

The subsequent loop consumes events from the FES and processes them. Events are processed in strict timestamp order to maintain causality, that is, to ensure that no current event may have an effect on earlier events.

Processing an event involves calls to user-supplied code. For example, using the computer network simulation example, processing a "timeout expired" event may consist of re-sending a copy of the network packet, updating the retry count, scheduling another "timeout" event, and so on. The user code may also remove events from the FES, for example when canceling timeouts.

The simulation stops when there are no events left (this rarely happens in practice), or when it isn't necessary for the simulation to run further because the model time or the CPU time has reached a given limit, or because the statistics have reached the desired accuracy. At this time, before the program exits, the user will typically want to record statistics into output files.

4.1.3 Events and Event Execution Order in OMNEST

OMNEST uses messages to represent events.[1] Messages are represented by instances of the `cMessage` class and its subclasses. Messages are sent from one module to another – this

[1] For all practical purposes. Note that there is a class called `cEvent` that `cMessage` subclasses from, but it is only used internal to the simulation kernel.

means that the place where the "event will occur" is the *message's destination module*, and the model time when the event occurs is the *arrival time* of the message. Events like "timeout expired" are implemented by the module sending a message to itself.

Events are consumed from the FES in arrival time order, to maintain causality. More precisely, given two messages, the following rules apply:

1. The message with the **earlier arrival time** is executed first. If arrival times are equal,
2. the one with the **higher scheduling priority** (smaller numeric value) is executed first. If priorities are the same,
3. the one **scheduled/sent earlier** is executed first.

Scheduling priority is a user-assigned integer attribute of messages.

4.1.4 Simulation Time

The current simulation time can be obtained with the `simTime()` function.

Simulation time in OMNEST is represented by the C++ type `simtime_t`, which is by default a typedef to the `SimTime` class. `SimTime` class stores simulation time in a 64-bit integer, using decimal fixed-point representation. The resolution is controlled by the *scale exponent* global configuration variable; that is, `SimTime` instances have the same resolution. The exponent can be chosen between -18 (attosecond resolution) and 0 (seconds). Some exponents with the ranges they provide are shown in the following table.

Exponent	Resolution	Approx. Range
-18	10^{-18}s (1as)	± 9.22s
-15	10^{-15}s (1fs)	± 153.72 minutes
-12	10^{-12}s (1ps)	± 106.75 days
-9	10^{-9}s (1ns)	± 292.27 years
-6	10^{-6}s (1us)	± 292271 years
-3	10^{-3}s (1ms)	$\pm 2.9227e8$ years
0	1s	$\pm 2.9227e11$ years

Note that although simulation time cannot be negative, it is still useful to be able to represent negative numbers, because they often arise during the evaluation of arithmetic expressions.

There is no implicit conversion from `SimTime` to `double`, mostly because it would conflict with overloaded arithmetic operations of `SimTime`; use the `dbl()` method of `SimTime` or the `SIMTIME_DBL()` macro to convert. To reduce the need for `dbl()`, several functions and methods have overloaded variants that directly accept `SimTime`, for example `fabs()`, `fmod()`, `div()`, `ceil()`, `floor()`, `uniform()`, `exponential()`, and `normal()`.

Other useful methods of `SimTime` include `str()`, which returns the value as a string; `parse()`, which converts a string to `SimTime`; `raw()`, which returns the underlying 64-bit integer; `getScaleExp()`, which returns the global scale exponent; `isZero()`, which tests whether the simulation time is 0; and `getMaxTime()`, which returns the maximum simulation time that can be represented at the current scale exponent. Zero and the maximum simulation time are also accessible via the `SIMTIME_ZERO` and `SIMTIME_MAX` macros.

```
// 340 microseconds in the future, truncated to millisecond boundary
simtime_t timeout = (simTime() + SimTime(340, SIMTIME_US)).trunc(SIMTIME_MS);
```

NOTE: Converting a `SimTime` to `double` may lose precision, because `double` only has a 52-bit mantissa. Earlier versions of OMNEST used `double` for the simulation time, but that caused problems in long simulations that relied on fine-grained timing, for example MAC protocols. Other problems were the accumulation of rounding errors, and non-associativity (often $(x+y)+z \neq x+(y+z)$, see [Gol91]) which meant that two `double` simulation times could not be reliably compared for equality.

4.1.5 FES Implementation

The implementation of the FES is a crucial factor in the performance of a discrete event simulator. In OMNEST, the FES is replaceable, and the default FES implementation uses *binary heap* as data structure. Binary heap is generally considered to be the best FES algorithm for discrete event simulation, as it provides a good, balanced performance for most workloads. (Exotic data structures like *skiplist* may perform better than heap in some cases.)

4.2 Components, Simple Modules, Channels

OMNEST simulation models are composed of modules and connections. Modules may be simple (atomic) modules or compound modules; simple modules are the active components in a model, and their behaviour is defined by the user as C++ code. Connections may have associated channel objects. Channel objects encapsulate channel behavior: propagation and transmission time modeling, error modeling, and possibly others. Channels are also programmable in C++ by the user.

Modules and channels are represented with the `cModule` and `cChannel` classes, respectively. `cModule` and `cChannel` are both derived from the `cComponent` class.

The user defines simple module types by subclassing `cSimpleModule`. Compound modules are instantiated with `cModule`, although the user can override it with `@class` in the NED file, and can even use a simple module C++ class (i.e. one derived from `cSimpleModule`) for a compound module.

The `cChannel`'s subclasses include the three built-in channel types: `cIdealChannel`, `cDelayChannel` and `cDatarateChannel`. The user can create new channel types by subclassing `cChannel` or any other channel class.

The following inheritance diagram illustrates the relationship of the classes mentioned above.

Simple modules and channels can be programmed by redefining certain member functions, and providing your own code in them. Some of those member functions are declared on `cComponent`, the common base class of channels and modules.

`cComponent` has the following member functions meant for redefining in subclasses:

- `initialize()`. This method is invoked after OMNEST has set up the network (i.e. created modules and connected them according to the definitions), and provides a place for initialization code;
- `finish()` is called when the simulation has terminated successfully, and its recommended use is the recording of summary statistics.

`initialize()` and `finish()`, together with `initialize()`'s variants for multi-stage initialization, will be covered in detail in section 4.3.3.

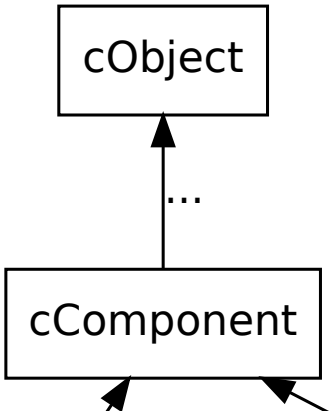


Figure 4.1: Inheritance of component, module and channel classes

In OMNEST, events occur inside simple modules. Simple modules encapsulate C++ code that generates events and reacts to events, implementing the behaviour of the module.

To define the dynamic behavior of a simple module, one of the following member functions need to be overridden:

- `handleMessage(cMessage *msg)`. It is invoked with the message as parameter whenever the module receives a message. `handleMessage()` is expected to *process* the message, and then return. Simulation time never elapses inside `handleMessage()` calls, only between them.

- `activity()` is started as a coroutine[2] at the beginning of the simulation, and it runs until the end of simulation (or until the function returns or otherwise terminates). Messages are obtained with `receive()` calls. Simulation time elapses inside `receive()` calls.

Modules written with `activity()` and `handleMessage()` can be freely mixed within a simulation model. Generally, `handleMessage()` should be preferred to `activity()`, due to scalability and other practical reasons. The two functions will be described in detail in sections 4.4.1 and 4.4.2, including their advantages and disadvantages.

The behavior of channels can also be modified by redefining member functions. However, the channel API is slightly more complicated than that of simple modules, so we'll describe it in a later section (4.8).

Last, let us mention `refreshDisplay()`, which is related to updating the visual appearance of the simulation when run under a graphical user interface. `refreshDisplay()` is covered in the chapter that deals with simulation visualization (8.2).

NOTE: `refreshDisplay()` has been added in OMNEST 5.0. Until then, visualization-related tasks were usually implemented as part of `handleMessage()`. `refreshDisplay()` provides a far superior and more efficient solution.

[2] Cooperatively scheduled thread, explained later.

4.3 Defining Simple Module Types

4.3.1 Overview

As mentioned before, a simple module is nothing more than a C++ class which has to be subclassed from `cSimpleModule`, with one or more virtual member functions redefined to define its behavior.

The class has to be registered with OMNEST via the `Define_Module()` macro. The `Define_Module()` line should always be put into `.cc` or `.cpp` files and not header file (`.h`), because the compiler generates code from it.

The following `HelloModule` is about the simplest simple module one could write. (We could have left out the `initialize()` method as well to make it even smaller, but how would it say Hello then?) Note `cSimpleModule` as base class, and the `Define_Module()` line.

```
// file: HelloModule.cc
#include <omnetpp.h>
using namespace omnetpp;

class HelloModule : public cSimpleModule
{
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

// register module class with OMNEST
Define_Module(HelloModule);

void HelloModule::initialize()
{
    EV << "Hello World!\n";
}

void HelloModule::handleMessage(cMessage *msg)
{
    delete msg; // just discard everything we receive
}
```

In order to be able to refer to this simple module type in NED files, we also need an associated NED declaration which might look like this:

```
// file: HelloModule.ned
simple HelloModule
{
    gates:
        input in;
}
```

4.3.2 Constructor

Simple modules are never instantiated by the user directly, but rather by the simulation kernel. This implies that one cannot write arbitrary constructors: the signature must be what is expected by the simulation kernel. Luckily, this contract is very simple: the constructor must be public, and must take no arguments:

```
  public:
    HelloModule();  // constructor takes no arguments
```

`cSimpleModule` itself has two constructors:

1. `cSimpleModule()` – one without arguments
2. `cSimpleModule(size_t stacksize)` – one that accepts the coroutine stack size

The first version should be used with `handleMessage()` simple modules, and the second one with `activity()` modules. (With the latter, the `activity()` method of the module class runs as a coroutine which needs a separate CPU stack, usually of 16..32K. This will be discussed in detail later.) Passing zero stack size to the latter constructor also selects `handleMessage()`.

Thus, the following constructor definitions are all OK, and select `handleMessage()` to be used with the module:

```
HelloModule::HelloModule() {...}
HelloModule::HelloModule() : cSimpleModule() {...}
```

It is also OK to omit the constructor altogether, because the compiler-generated one is suitable too.

The following constructor definition selects `activity()` to be used with the module, with 16K of coroutine stack:

```
HelloModule::HelloModule() : cSimpleModule(16384) {...}
```

NOTE: The `Module_Class_Members()` macro, already deprecated in OMNEST 3.2, has been removed in the 4.0 version. When porting older simulation models, occurrences of this macro can simply be removed from the source code.

4.3.3 Initialization and Finalization

Basic Usage

The `initialize()` and `finish()` methods are declared as part of `cComponent`, and provide the user the opportunity of running code at the beginning and at successful termination of the simulation.

The reason `initialize()` exists is that usually you cannot put simulation-related code into the simple module constructor, because the simulation model is still being setup when the constructor runs, and many required objects are not yet available. In contrast, `initialize()` is called just before the simulation starts executing, when everything else has been set up already.

`finish()` is for recording statistics, and it only gets called when the simulation has terminated normally. It does not get called when the simulations stops with an error message. The

destructor always gets called at the end, no matter how the simulation stopped, but at that time it is fair to assume that the simulation model has been halfway demolished already.

Based on the above considerations, the following usage conventions exist for these four methods:

Constructor:

Set pointer members of the module class to `nullptr`; postpone all other initialization tasks to `initialize()`.

`initialize()`:

Perform all initialization tasks: read module parameters, initialize class variables, allocate dynamic data structures with `new`; also allocate and initialize self-messages (timers) if needed.

`finish()`:

Record statistics. Do **not** `delete` anything or cancel timers – all cleanup must be done in the destructor.

Destructor:

Delete everything which was allocated by `new` and is still held by the module class. With self-messages (timers), use the `cancelAndDelete(msg)` function! It is almost always wrong to just delete a self-message from the destructor, because it might be in the scheduled events list. The `cancelAndDelete(msg)` function checks for that first, and cancels the message before deletion if necessary.

OMNEST prints the list of unreleased objects at the end of the simulation. When a simulation model dumps *"undisposed object ..."* messages, this indicates that the corresponding module destructors should be fixed. As a temporary measure, these messages may be hidden by setting `print-undisposed=false` in the configuration.

NOTE: The `perform-gc` configuration option has been removed in OMNEST 4.0. Automatic garbage collection cannot be implemented reliably, due to the limitations of the C++ language.

Invocation Order

The `initialize()` functions of the modules are invoked *before* the first event is processed, but *after* the initial events (starter messages) have been placed into the FES by the simulation kernel.

Both simple and compound modules have `initialize()` functions. A compound module's `initialize()` function runs *before* that of its submodules.

The `finish()` functions are called when the event loop has terminated, and only if it terminated normally.

NOTE: `finish()` is not called if the simulation has terminated with a runtime error.

The calling order for `finish()` is the reverse of the order of `initialize()`: first submodules, then the encompassing compound module. [3]

[3] The way you can provide an `initialize()` function for a compound module is to subclass `cModule`, and tell OMNEST to use the new class for the compound module. The latter is done by adding the `@class(<classname>)` property into the NED declaration.

This is summarized in the following pseudocode:

```
perform simulation run:
    build network
      (i.e. the system module and its submodules recursively)
    insert starter messages for all submodules using activity()
    do callInitialize() on system module
        enter event loop // (described earlier)
    if (event loop terminated normally) // i.e. no errors
        do callFinish() on system module
    clean up

callInitialize()
{
    call to user-defined initialize() function
    if (module is compound)
        for (each submodule)
            do callInitialize() on submodule
}

callFinish()
{
    if (module is compound)
        for (each submodule)
            do callFinish() on submodule
    call to user-defined finish() function
}
```

Keep in mind that `finish()` is not always called, so it isn't a good place for cleanup code which should run every time the module is deleted. `finish()` is only a good place for writing statistics, result post-processing and other operations which are supposed to run only on successful completion. Cleanup code should go into the destructor.

Multi-Stage Initialization

In simulation models where one-stage initialization provided by `initialize()` is not sufficient, one can use multi-stage initialization. Modules have two functions which can be redefined by the user:

```
virtual void initialize(int stage);
virtual int numInitStages() const;
```

At the beginning of the simulation, `initialize(0)` is called for *all* modules, then `initialize(1)`, `initialize(2)`, etc. You can think of it like initialization takes place in several "waves". For each module, `numInitStages()` must be redefined to return the number of init stages required, e.g. for a two-stage init, `numInitStages()` should return 2, and `initialize(int stage)` must be implemented to handle the *stage=0* and *stage=1* cases. [4]

The `callInitialize()` function performs the full multi-stage initialization for that module and all its submodules.

[4] Note the `const` in the `numInitStages()` declaration. If you forget it, by C++ rules you create a *different* function instead of redefining the existing one in the base class, thus the existing one will remain in effect and return 1.

If you do not redefine the multi-stage initialization functions, the default behavior is single-stage initialization: the default `numInitStages()` returns 1, and the default `initialize(int stage)` simply calls `initialize()`.

"End-of-Simulation" Event

The task of `finish()` is implemented in several other simulators by introducing a special *end-of-simulation* event. This is not a very good practice because the simulation programmer has to code the models (often represented as FSMs) so that they can *always* properly respond to end-of-simulation events, in whichever state they are. This often makes program code unnecessarily complicated. For this reason OMNEST does not use the end of simulation event.

This can also be witnessed in the design of the PARSEC simulation language (UCLA). Its predecessor Maisie used end-of-simulation events, but – as documented in the PARSEC manual – this has led to awkward programming in many cases, so for PARSEC end-of-simulation events were dropped in favour of `finish()` (called `finalize()` in PARSEC).

4.4 Adding Functionality to cSimpleModule

This section discusses `cSimpleModule`'s previously mentioned `handleMessage()` and `activity()` member functions, intended to be redefined by the user.

4.4.1 handleMessage()

Function Called for Each Event

The idea is that at each event (message arrival) we simply call a user-defined function. This function, `handleMessage(cMessage *msg)` is a virtual member function of `cSimpleModule` which does nothing by default – the user has to redefine it in subclasses and add the message processing code.

The `handleMessage()` function will be called for every message that arrives at the module. The function should process the message and return immediately after that. The simulation time is potentially different in each call. No simulation time elapses within a call to `handleMessage()`.

The event loop inside the simulator handles both `activity()` and `handleMessage()` simple modules, and it corresponds to the following pseudocode:

```
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    m:= module containing this event
    if (m works with handleMessage())
        m->handleMessage( event )
    else // m works with activity()
        transferTo( m )
}
```

Modules with `handleMessage()` are NOT started automatically: the simulation kernel creates starter messages only for modules with `activity()`. This means that you have to schedule self-messages from the `initialize()` function if you want a `handleMessage()` simple module to start working "by itself", without first receiving a message from other modules.

Programming with handleMessage()

To use the `handleMessage()` mechanism in a simple module, you must specify *zero stack size* for the module. This is important, because this tells OMNEST that you want to use `handleMessage()` and not `activity()`.

Message/event related functions you can use in `handleMessage()`:

- `send()` family of functions – to send messages to other modules
- `scheduleAt()` – to schedule an event (the module "sends a message to itself")
- `cancelEvent()` – to delete an event scheduled with `scheduleAt()`

The `receive()` and `wait()` functions cannot be used in `handleMessage()`, because they are coroutine-based by nature, as explained in the section about `activity()`.

You have to add data members to the module class for every piece of information you want to preserve. This information cannot be stored in local variables of `handleMessage()` because they are destroyed when the function returns. Also, they cannot be stored in static variables in the function (or the class), because they would be shared between all instances of the class.

Data members to be added to the module class will typically include things like:

- state (e.g. IDLE/BUSY, CONN_DOWN/CONN_ALIVE/...)
- other variables which belong to the state of the module: retry counts, packet queues, etc.
- values retrieved/computed once and then stored: values of module parameters, gate indices, routing information, etc.
- pointers of message objects created once and then reused for timers, timeouts, etc.
- variables/objects for statistics collection

These variables are often initialized from the `initialize()` method, because the information needed to obtain the initial value (e.g. module parameters) may not yet be available at the time the module constructor runs.

Another task to be done in `initialize()` is to schedule initial event(s) which trigger the first call(s) to `handleMessage()`. After the first call, `handleMessage()` must take care to schedule further events for itself so that the "chain" is not broken. Scheduling events is not necessary if your module only has to react to messages coming from other modules.

`finish()` is normally used to record statistics information accumulated in data members of the class at the end of the simulation.

Application Area

`handleMessage()` is in most cases a better choice than `activity()`:

1. When you expect the module to be used in large simulations, involving several thousand modules. In such cases, the module stacks required by `activity()` would simply consume too much memory.

2. For modules which maintain little or no state information, such as packet sinks, `handleMessage()` is more convenient to program.

3. Other good candidates are modules with a large state space and many arbitrary state transition possibilities (i.e. where there are many possible subsequent states for any state). Such algorithms are difficult to program with `activity()`, and better suited for `handleMessage()` (see rule of thumb below). This is the case for most communication protocols.

Example 1: Protocol Models

Models of protocol layers in a communication network tend to have a common structure on a high level because fundamentally they all have to react to three types of events: to messages arriving from higher layer protocols (or apps), to messages arriving from lower layer protocols (from the network), and to various timers and timeouts (that is, self-messages).

This usually results in the following source code pattern:

```
class FooProtocol : public cSimpleModule
{
  protected:
    // state variables
    // ...

    virtual void processMsgFromHigherLayer(cMessage *packet);
    virtual void processMsgFromLowerLayer(FooPacket *packet);
    virtual void processTimer(cMessage *timer);

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

// ...

void FooProtocol::handleMessage(cMessage *msg)
{
    if (msg->isSelfMessage())
        processTimer(msg);
    else if (msg->arrivedOn("fromNetw"))
        processMsgFromLowerLayer(check_and_cast<FooPacket *>(msg));
    else
        processMsgFromHigherLayer(msg);
}
```

The functions `processMsgFromHigherLayer()`, `processMsgFromLowerLayer()` and `processTimer()` are then usually split further: there are separate methods to process separate packet types and separate timers.

Example 2: Simple Traffic Generators and Sinks

The code for simple packet generators and sinks programmed with `handleMessage()` might be as simple as the following pseudocode:

```
PacketGenerator::handleMessage(msg)
{
    create and send out a new packet;
    schedule msg again to trigger next call to handleMessage;
}

PacketSink::handleMessage(msg)
{
    delete msg;
}
```

Note that *PacketGenerator* will need to redefine `initialize()` to create *m* and schedule the first event.

The following simple module generates packets with exponential inter-arrival time. (Some details in the source haven't been discussed yet, but the code is probably understandable nevertheless.)

```
class Generator : public cSimpleModule
{
  public:
    Generator() : cSimpleModule() {}
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(Generator);

void Generator::initialize()
{
    // schedule first sending
    scheduleAt(simTime(), new cMessage);
}

void Generator::handleMessage(cMessage *msg)
{
    // generate & send packet
    cMessage *pkt = new cMessage;
    send(pkt, "out");
    // schedule next call
    scheduleAt(simTime()+exponential(1.0), msg);
}
```

Example 3: Bursty Traffic Generator

A bit more realistic example is to rewrite our Generator to create packet bursts, each consisting of `burstLength` packets.

We add some data members to the class:

- `burstLength` will store the parameter that specifies how many packets a burst must contain,
- `burstCounter` will count in how many packets are left to be sent in the current burst.

The code:

```
class BurstyGenerator : public cSimpleModule
{
  protected:
    int burstLength;
    int burstCounter;

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(BurstyGenerator);

void BurstyGenerator::initialize()
{
    // init parameters and state variables
    burstLength = par("burstLength");
    burstCounter = burstLength;
    // schedule first packet of first burst
    scheduleAt(simTime(), new cMessage);
}

void BurstyGenerator::handleMessage(cMessage *msg)
{
    // generate & send packet
    cMessage *pkt = new cMessage;
    send(pkt, "out");
    // if this was the last packet of the burst
    if (--burstCounter == 0) {
        // schedule next burst
        burstCounter = burstLength;
        scheduleAt(simTime()+exponential(5.0), msg);
    }
    else {
        // schedule next sending within burst
        scheduleAt(simTime()+exponential(1.0), msg);
    }
}
```

Pros and Cons of Using `handleMessage()`

Pros:

- consumes less memory: no separate stack needed for simple modules
- fast: function call is faster than switching between coroutines

Cons:

- local variables cannot be used to store state information
- need to redefine `initialize()`

Usually, `handleMessage()` should be preferred over `activity()`.

Other Simulators

Many simulation packages use a similar approach, often topped with something like a state machine (FSM) which hides the underlying function calls. Such systems are:

- OPNET^{TM} which uses FSM's designed using a graphical editor;
- NetSim++ clones OPNET's approach;
- SMURPH (University of Alberta) defines a (somewhat eclectic) language to describe FSMs, and uses a precompiler to turn it into C++ code;
- Ptolemy (UC Berkeley) uses a similar method.

OMNEST's FSM support is described in the next section.

4.4.2 activity()

Process-Style Description

With `activity()`, a simple module can be coded much like an operating system process or thread. One can wait for an incoming message (event) at any point of the code, suspend the execution for some time (model time!), etc. When the `activity()` function exits, the module is terminated. (The simulation can continue if there are other modules which can run.)

The most important functions that can be used in `activity()` are (they will be discussed in detail later):

- `receive()` – to receive messages (events)
- `wait()` – to suspend execution for some time (model time)
- `send()` family of functions – to send messages to other modules
- `scheduleAt()` – to schedule an event (the module "sends a message to itself")
- `cancelEvent()` – to delete an event scheduled with `scheduleAt()`
- `end()` – to finish execution of this module (same as exiting the `activity()` function)

The `activity()` function normally contains an infinite loop, with at least a `wait()` or `receive()` call in its body.

Application Area

Generally you should prefer `handleMessage()` to `activity()`. The main problem with `activity()` is that it doesn't scale because every module needs a separate coroutine stack. It has also been observed that `activity()` does not encourage a good programming style, and stack switching also confuses many debuggers.

There is one scenario where `activity()`'s process-style description is convenient: when the process has many states but transitions are very limited, i.e. from any state the process can only go to one or two other states. For example, this is the case when programming a network application, which uses a single network connection. The pseudocode of the application which talks to a transport layer protocol might look like this:

```
activity()
{
    while(true)
    {
        open connection by sending OPEN command to transport layer
        receive reply from transport layer
        if (open not successful)
        {
            wait(some time)
            continue // loop back to while()
        }

        while (there is more to do)
        {
            send data on network connection
            if (connection broken)
            {
                continue outer loop // loop back to outer while()
            }
            wait(some time)
            receive data on network connection
            if (connection broken)
            {
                continue outer loop // loop back to outer while()
            }
            wait(some time)
        }
        close connection by sending CLOSE command to transport layer
        if (close not successful)
        {
            // handle error
        }
        wait(some time)
    }
}
```

If there is a need to handle several connections concurrently, dynamically creating simple modules to handle each is an option. Dynamic module creation will be discussed later.

There are situations when you certainly *do not want* to use `activity()`. If the `activity()`

function contains no `wait()` and it has only one `receive()` at the top of a message handling loop, there is no point in using `activity()`, and the code should be written with `handleMessage()`. The body of the loop would then become the body of `handleMessage()`, state variables inside `activity()` would become data members in the module class, and they would be initialized in `initialize()`.

Example:

```
void Sink::activity()
{
    while(true) {
        msg = receive();
        delete msg;
    }
}
```

should rather be programmed as:

```
void Sink::handleMessage(cMessage *msg)
{
    delete msg;
}
```

Activity() Is Run as a Coroutine

`activity()` is run in a coroutine. Coroutines are similar to threads, but are scheduled non-preemptively (this is also called cooperative multitasking). One can switch from one coroutine to another coroutine by a `transferTo(otherCoroutine)` call, causing the first coroutine to be suspended and second one to run. Later, when the second coroutine performs a `transferTo(firstCoroutine)` call to the first one, the execution of the first coroutine will resume from the point of the `transferTo(otherCoroutine)` call. The full state of the coroutine, including local variables are preserved while the thread of execution is in other coroutines. This implies that each coroutine has its own CPU stack, and `transferTo()` involves a switch from one CPU stack to another.

Coroutines are at the heart of OMNEST, and the simulation programmer doesn't ever need to call `transferTo()` or other functions in the coroutine library, nor does he need to care about the coroutine library implementation. It is important to understand, however, how the event loop found in discrete event simulators works with coroutines.

When using coroutines, the event loop looks like this (simplified):

```
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    transferTo(module containing the event)
}
```

That is, when a module has an event, the simulation kernel transfers the control to the module's coroutine. It is expected that when the module "decides it has finished the processing of the event", it will transfer the control back to the simulation kernel by a `transferTo(main)` call. Initially, simple modules using `activity()` are "booted" by events ("*starter messages*") inserted into the FES by the simulation kernel before the start of the simulation.

How does the coroutine know it has "finished processing the event"? The answer: *when it requests another event*. The functions which request events from the simulation kernel are the `receive()` and `wait()`, so their implementations contain a `transferTo(main)` call somewhere.

Their pseudocode, as implemented in OMNEST:

```
receive()
{
    transferTo(main)
    retrieve current event
    return the event // remember: events = messages
}

wait()
{
    create event e
    schedule it at (current sim. time + wait interval)
    transferTo(main)
    retrieve current event
    if (current event is not e) {
        error
    }
    delete e  // note: actual impl. reuses events
    return
}
```

Thus, the `receive()` and `wait()` calls are special points in the `activity()` function, because they are where

- simulation time elapses in the module, and
- other modules get a chance to execute.

Starter Messages

Modules written with `activity()` need starter messages to "boot". These starter messages are inserted into the FES automatically by OMNEST at the beginning of the simulation, even before the `initialize()` functions are called.

Coroutine Stack Size

The simulation programmer needs to define the CPU stack size for coroutines. This cannot be automated.

16 or 32 kbytes is usually a good choice, but more space may be needed if the module uses recursive functions or has many/large local variables. OMNEST has a built-in mechanism that will usually detect if the module stack is too small and overflows. OMNEST can also report how much stack space a module actually uses at runtime.

initialize() and finish() with activity()

Because local variables of `activity()` are preserved across events, you can store everything (state information, packet buffers, etc.) in them. Local variables can be initialized at the top of the `activity()` function, so there isn't much need to use `initialize()`.

You do need `finish()`, however, if you want to write statistics at the end of the simulation. Because `finish()` cannot access the local variables of `activity()`, you have to put the variables and objects containing the statistics into the module class. You still don't need `initialize()` because class members can also be initialized at the top of `activity()`.

Thus, a typical setup looks like this in pseudocode:

```
class MySimpleModule...
{
    ...
    variables for statistics collection
    activity();
    finish();
};

MySimpleModule::activity()
{
    declare local vars and initialize them
    initialize statistics collection variables

    while(true)
    {
        ...
    }
}

MySimpleModule::finish()
{
    record statistics into file
}
```

Pros and Cons of Using `activity()`

Pros:

- `initialize()` not needed, state can be stored in local variables of `activity()`
- process-style description is a natural programming model in some cases

Cons:

- limited scalability: coroutine stacks can unacceptably increase the memory requirements of the simulation program if there are many `activity()`-based simple modules;
- run-time overhead: switching between coroutines is slower than a simple function call
- does not encourage a good programming style: as module complexity grows, `activity()` tends to become a large, monolythic function.

In most cases, cons outweigh pros and it is a better idea to use `handleMessage()` instead.

Other Simulators

Coroutines are used by a number of other simulation packages:

- All simulation software which inherits from SIMULA (e.g. C++SIM) is based on coroutines, although all in all the programming model is quite different.
- The simulation/parallel programming language Maisie and its successor PARSEC (from UCLA) also use coroutines (although implemented with "normal" preemptive threads). The philosophy is quite similar to OMNEST. PARSEC, being "just" a programming language, it has a more elegant syntax but far fewer features than OMNEST.
- Many Java-based simulation libraries are based on Java threads.

4.4.3 Use Modules Instead of Global Variables

If possible, avoid using global variables, including static class members. They are prone to cause several problems. First, they are not reset to their initial values (to zero) when you rebuild the simulation in Qtenv, or start another run in Cmdenv. This may produce surprising results. Second, they prevent you from parallelizing the simulation. When using parallel simulation, each partition of the model runs in a separate process, having their own copies of global variables. This is usually not what you want.

The solution is to encapsulate the variables into simple modules as private or protected data members, and expose them via public methods. Other modules can then call these public methods to get or set the values. Calling methods of other modules will be discussed in section 4.12. Examples of such modules are `InterfaceTable` and `RoutingTable` in *INET Framework*.

4.4.4 Reusing Module Code via Subclassing

The code of simple modules can be reused via subclassing, and redefining virtual member functions. An example:

```
class TransportProtocolExt : public TransportProtocol
{
  protected:
    virtual void recalculateTimeout();
};

Define_Module(TransportProtocolExt);

void TransportProtocolExt::recalculateTimeout()
{
    //...
}
```

The corresponding NED declaration:

```
simple TransportProtocolExt extends TransportProtocol
{
    @class(TransportProtocolExt);  // Important!
}
```

NOTE: Note the `@class()` property, which tells OMNEST to use the `TransportProtocolExt` C++ class for the module type! It is needed because NED inheritance is NED inheritance *only*, so without `@class()` the `TransportProtocolExt` NED type would inherit the C++ class from its base NED type.

4.5 Accessing Module Parameters

Module parameters declared in NED files are represented with the `cPar` class at runtime, and be accessed by calling the `par()` member function of `cComponent`:

```
cPar& delayPar = par("delay");
```

`cPar`'s value can be read with methods that correspond to the parameter's NED type: `boolValue()`, `intValue()`, `doubleValue()`, `stringValue()`, `stdstringValue()`, `xmlValue()`. There are also overloaded type cast operators for the corresponding types (`bool`; integer types including `int`, `long`, etc; `double`; `const char *`; `cXMLElement *`).

```
long numJobs = par("numJobs").intValue();
double processingDelay = par("processingDelay"); // using operator double()
```

Note that `cPar` has two methods for returning a string value: `stringValue()`, which returns `const char *`, and `stdstringValue()`, which returns `std::string`. For volatile parameters, only `stdstringValue()` may be used, but otherwise the two are interchangeable.

If you use the `par("foo")` parameter in expressions (such as `4*par("foo")+2`), the C++ compiler may be unable to decide between overloaded operators and report ambiguity. This issue can be resolved by adding an explicit cast such as `(double)par("foo")`, or using the `doubleValue()` or `intValue()` methods.

4.5.1 Volatile and Non-Volatile Parameters

A parameter can be declared `volatile` in the NED file. The `volatile` modifier indicates that a parameter is re-read every time a value is needed during simulation. Volatile parameters typically are used for things like random packet generation interval, and are assigned values like `exponential(1.0)` (numbers drawn from the exponential distribution with mean 1.0).

In contrast, non-volatile NED parameters are constants, and reading their values multiple times is guaranteed to yield the same value. When a non-volatile parameter is assigned a random value like `exponential(1.0)`, it is evaluated once at the beginning of the simulation and replaced with the result, so all reads will get same (randomly generated) value.

The typical usage for non-volatile parameters is to read them in the `initialize()` method of the module class, and store the values in class variables for easy access later:

```
class Source : public cSimpleModule
{
  protected:
```

```
    long numJobs;
    virtual void initialize();
    ...
};

void Source::initialize()
{
    numJobs = par("numJobs");
    ...
}
```

`volatile` parameters need to be re-read every time the value is needed. For example, a parameter that represents a random packet generation interval may be used like this:

```
void Source::handleMessage(cMessage *msg)
{
    ...
    scheduleAt(simTime() + par("interval").doubleValue(), timerMsg);
    ...
}
```

This code looks up the the parameter by name every time. This lookup can be avoided by storing the parameter object's pointer in a class variable, resulting in the following code:

```
class Source : public cSimpleModule
{
  protected:
    cPar *intervalp;
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    ...
};

void Source::initialize()
{
    intervalp = &par("interval");
    ...
}

void Source::handleMessage(cMessage *msg)
{
    ...
    scheduleAt(simTime() + intervalp->doubleValue(), timerMsg);
    ...
}
```

4.5.2 Changing a Parameter's Value

Parameter values can be changed from the program, during execution. This is rarely needed, but may be useful for some scenarios.

NOTE: The parameter's type cannot be changed at runtime – it must remain the type declared in the NED file. It is also not possible to add or remove module parameters at runtime.

The methods to set the parameter value are `setBoolValue()`, `setLongValue()`, `setStringValue()`, `setDoubleValue()`, `setXMLValue()`. There are also overloaded assignment operators for various types including `bool`, `int`, `long`, `double`, `const char *`, and `cXMLElement *`.

To allow a module to be notified about parameter changes, override its `handleParameterChange()` method, see 4.5.5.

4.5.3 Further cPar Methods

The parameter's name and type are returned by the `getName()` and `getType()` methods. The latter returns a value from an enum, which can be converted to a readable string with the `getTypeName()` static method. The enum values are `BOOL`, `DOUBLE`, `LONG`, `STRING` and `XML`; and since the enum is an inner type, they usually have to be qualified with `cPar::`.

`isVolatile()` returns whether the parameter was declared volatile in the NED file. `isNumeric()` returns true if the parameter type is double or long.

The `str()` method returns the parameter's value in a string form. If the parameter contains an expression, then the string representation of the expression is returned.

An example usage of the above methods:

```
int n = getNumParams();
for (int i = 0; i < n; i++)
{
    cPar& p = par(i);
    EV << "parameter: " << p.getName() << "\n";
    EV << "  type:" << cPar::getTypeName(p.getType()) << "\n";
    EV << "  contains:" << p.str() << "\n";
}
```

The NED properties of a parameter can be accessed with the `getProperties()` method that returns a pointer to the `cProperties` object that stores the properties of this parameter. Specifically, `getUnit()` returns the unit of measurement associated with the parameter (`@unit` property in NED).

Further `cPar` methods and related classes like `cExpression` and `cDynamicExpression` are used by the NED infrastructure to set up and assign parameters. They are documented in the **API Reference**, but they are normally of little interest to users.

4.5.4 Object Parameters

As of version 4.2, OMNEST does not support parameter arrays, but in practice they can be emulated using string parameters. One can assign the parameter a string which contains all values in a textual form (for example, `"0 1.234 3.95 5.467"`), then parse this string in the simple module.

The `cStringTokenizer` class can be quite useful for this purpose. The constructor accepts a string, which it regards as a sequence of tokens (words) separated by delimiter characters

(by default, spaces). Then you can either enumerate the tokens and process them one by one (`hasMoreTokens()`, `nextToken()`), or use one of the `cStringTokenizer` convenience methods to convert them into a vector of strings (`asVector()`), integers (`asIntVector()`), or `double`s (`asDoubleVector()`).

The latter methods can be used like this:

```
const char *vstr = par("v").stringValue(); // e.g. "aa bb cc";
std::vector<std::string> v = cStringTokenizer(vstr).asVector();
```

and

```
const char *str = "34 42 13 46 72 41";
std::vector<int> v = cStringTokenizer().asIntVector();

const char *str = "0.4311 0.7402 0.7134";
std::vector<double> v = cStringTokenizer().asDoubleVector();
```

The following example processes the string by enumerating the tokens:

```
const char *str = "3.25 1.83 34 X 19.8"; // input

std::vector<double> result;
cStringTokenizer tokenizer(str);
while (tokenizer.hasMoreTokens())
{
    const char *token = tokenizer.nextToken();
    if (strcmp(token, "X")==0)
        result.push_back(DEFAULT_VALUE);
    else
        result.push_back(atof(token));
}
```

4.5.5 handleParameterChange()

It is possible for modules to be notified when the value of a parameter changes at runtime, possibly due to another module dynamically changing it. The typical action is to re-read the parameter, and update the module's state if needed.

To enable notification, redefine the `handleParameterChange()` method of the module class. This method will be called back by the simulation kernel with the parameter name as argument every time a new value is assigned to a parameter. The method signature is the following:

```
void handleParameterChange(const char *parameterName);
```

The following example shows a module that re-reads its `serviceTime` parameter when its value changes:

```
void Queue::handleParameterChange(const char *parameterName)
{
    if (strcmp(parameterName, "serviceTime") == 0)
        serviceTime = par("serviceTime"); // refresh data member
}
```

NOTE: As a rule of thumb, `handleParameterChange()` should handle all parameters that are marked as `@mutable` in the NED file.

Notifications are suppressed while the network (or module) is being set up.[5]

`handleParameterChange()` methods need to be implemented carefully, because they may be called at a time when the module has not yet completed all initialization stages.

Also, be extremely careful when changing parameters from inside `handleParameterChange()`, because it is easy to accidentally create an infinite notification loop.

4.6 Accessing Gates and Connections

4.6.1 Gate Objects

Module gates are represented by `cGate` objects. Gate objects know to which other gates they are connected, and what are the channel objects associated with the links.

Accessing Gates by Name

The `cModule` class has a number of member functions that deal with gates. You can look up a gate by name using the `gate()` method:

```
cGate *outGate = gate("out");
```

This works for input and output gates. However, when a gate was declared `inout` in NED, it is actually represented by the simulation kernel with two gates, so the above call would result in a *gate not found* error. The `gate()` method needs to be told whether the input or the output half of the gate you need. This can be done by appending the `"$i"` or `"$o"` to the gate name. The following example retrieves the two gates for the inout gate `"g"`:

```
cGate *gIn = gate("g$i");
cGate *gOut = gate("g$o");
```

Another way is to use the `gateHalf()` function, which takes the inout gate's name plus either `cGate::INPUT` or `cGate::OUTPUT`:

```
cGate *gIn = gateHalf("g", cGate::INPUT);
cGate *gOut = gateHalf("g", cGate::OUTPUT);
```

These methods throw an error if the gate does not exist, so they cannot be used to determine whether the module has a particular gate. For that purpose there is a `hasGate()` method. An example:

```
if (hasGate("optOut"))
   send(new cMessage(), "optOut");
```

A gate can also be identified and looked up by a numeric gate ID. You can get the ID from the gate itself (`getId()` method), or from the module by gate name (`findGate()` method). The `gate()` method also has an overloaded variant which returns the gate from the gate ID.

[5] Prior to OMNEST 6.0, notifications were also disabled during the initialization phase (see 4.3.3), and additionally, a `handleParameterChange(nullptr)` call was made by the simulation kernel after the last stage of initialization. They are no longer done, and simulation models exploiting the previous behavior need to be updated.

```
int gateId = gate("in")->getId();  // or:
int gateId = findGate("in");
```

As gate IDs are more useful with gate vectors, we'll cover them in detail in a later section.

Gate Vectors

Gate vectors possess one `cGate` object per element. To access individual gates in the vector, you need to call the `gate()` function with an additional *index* parameter. The index should be between zero and *size*-1. The size of the gate vector can be read with the `gateSize()` method. The following example iterates through all elements in the gate vector:

```
for (int i = 0; i < gateSize("out"); i++) {
    cGate *gate = gate("out", i);
    //...
}
```

A gate vector cannot have "holes" in it; that is, `gate()` never returns `nullptr` or throws an error if the gate vector exists and the index is within bounds.

For inout gates, `gateSize()` may be called with or without the `"$i"/"$o"` suffix, and returns the same number.

The `hasGate()` method may be used both with and without an index, and they mean two different things: without an index it tells the existence of a gate vector with the given name, regardless of its size (it returns `true` for an existing vector even if its size is currently zero!); with an index it also examines whether the index is within the bounds.

Gate IDs

A gate can also be accessed by its ID. A very important property of gate IDs is that they are *contiguous* within a gate vector, that is, the ID of a gate `g[k]` can be calculated as the ID of `g[0]` plus k. This allows you to efficiently access any gate in a gate vector, because retrieving a gate by ID is more efficient than by name and index. The index of the first gate can be obtained with `gate("out",0)->getId()`, but it is better to use a dedicated method, `gateBaseId()`, because it also works when the gate vector size is zero.

Two further important properties of gate IDs: they are *stable* and *unique* (within the module). By stable we mean that the ID of a gate never changes; and by unique we not only mean that at any given time no two gates have the same IDs, but also that IDs of deleted gates do not get reused later, so gate IDs are unique in the lifetime of a simulation run.

> **NOTE:** OMNEST version earlier than 4.0 did not have these guarantees – resizing a gate vector could cause its ID range to be relocated, if it would have overlapped with the ID range of other gate vectors. OMNEST 4.x solves the same problem by interpreting the gate ID as a bitfield, basically containing bits that identify the gate name, and other bits that hold the index. This also means that the theoretical upper limit for a gate size is now smaller, albeit it is still big enough so that it can be safely ignored for practical purposes.

The following example iterates through a gate vector, using IDs:

```
int baseId = gateBaseId("out");
int size = gateSize("out");
```

```
for (int i = 0; i < size; i++) {
    cGate *gate = gate(baseId + i);
    //...
}
```

Enumerating All Gates

If you need to go through all gates of a module, there are two possibilities. One is invoking the `getGateNames()` method that returns the names of all gates and gate vectors the module has; then you can call `isGateVector(name)` to determine whether individual names identify a scalar gate or a gate vector; then gate vectors can be enumerated by index. Also, for inout gates `getGateNames()` returns the base name without the `"$i"/"$o"` suffix, so the two directions need to be handled separately. The `gateType(name)` method can be used to test whether a gate is inout, input or output (it returns `cGate::INOUT`, `cGate::INPUT`, or `cGate::OUTPUT`).

Clearly, the above solution can be quite difficult. An alternative is to use the `GateIterator` class provided by `cModule`. It goes like this:

```
for (cModule::GateIterator i(this); !i.end(); i++) {
    cGate *gate = *i;
    ...
}
```

Where `this` denotes the module whose gates are being enumerated (it can be replaced by any `cModule *` variable).

> **NOTE:** In earlier OMNEST versions, gate IDs used to be small integers, so it made sense to iterate over all gates of a module by enumerating all IDs from zero to a maximum, skipping the holes (`nullptr`s). This is no longer the case with OMNEST 4.0 and later versions. Moreover, the `gate()` method now throws an error when called with an invalid ID, and not just returns `nullptr`.

Adding and Deleting Gates

Although rarely needed, it is possible to add and remove gates during simulation. You can add scalar gates and gate vectors, change the size of gate vectors, and remove scalar gates and whole gate vectors. It is not possible to remove individual random gates from a gate vector, to remove one half of an inout gate (e.g. `"gate$o"`), or to set different gate vector sizes on the two halves of an inout gate vector.

The `cModule` methods for adding and removing gates are `addGate(name,type,isvector=false)` and `deleteGate(name)`. Gate vector size can be changed by using `setGateSize(name,size)`. None of these methods accept `"$i"` / `"$o"` suffix in gate names.

> **NOTE:** When memory efficiency is of concern, it is useful to know that in OMNEST 4.0 and later, a gate vector will consume significantly less memory than the same number of individual scalar gates.

cGate Methods

The `getName()` method of `cGate` returns the name of the gate or gate vector without the index. If you need a string that contains the gate index as well, `getFullName()` is what you want. If you also want to include the hierarchical name of the owner module, call `getFullPath()`.

The `getType()` method of `cGate` returns the gate type, either `cGate::INPUT` or `cGate::OUTPUT`. (It cannot return `cGate::INOUT`, because an inout gate is represented by a pair of `cGate`s.)

If you have a gate that represents half of an inout gate (that is, `getName()` returns something like `"g$i"` or `"g$o"`), you can split the name with the `getBaseName()` and `getNameSuffix()` methods. `getBaseName()` method returns the name without the `$i`/`$o` suffix; and `getNameSuffix()` returns just the suffix (including the dollar sign). For normal gates, `getBaseName()` is the same as `getName()`, and `getNameSuffix()` returns the empty string.

The `isVector()`, `getIndex()`, `getVectorSize()` speak for themselves; `size()` is an alias to `getVectorSize()`. For non-vector gates, `getIndex()` returns 0 and `getVectorSize()` returns 1.

The `getId()` method returns the gate ID (not to be confused with the gate index).

The `getOwnerModule()` method returns the module the gate object belongs to.

To illustrate these methods, we expand the gate iterator example to print some information about each gate:

```
for (cModule::GateIterator i(this); !i.end(); i++) {
    cGate *gate = *i;
    EV << gate->getFullName() << ": ";
    EV << "id=" << gate->getId() << ", ";
    if (!gate->isVector())
        EV << "scalar gate, ";
    else
        EV << "gate " << gate->getIndex()
           << " in vector " << gate->getName()
           << " of size " << gate->getVectorSize() << ", ";
    EV << "type:" << cGate::getTypeName(gate->getType());
    EV << "\n";
}
```

There are further `cGate` methods to access and manipulate the connection(s) attached to the gate; they will be covered in the following sections.

4.6.2 Connections

Simple module gates have normally one connection attached. Compound module gates, however, need to be connected both inside and outside of the module to be useful. A series of connections (joined with compound module gates) is called a *connection path* or just path. A path is directed, and it normally starts at an output gate of a simple module, ends at an input gate of a simple module, and passes through several compound module gates.

Every `cGate` object contains pointers to the previous gate and the next gate in the path (returned by the `getPreviousGate()` and `getNextGate()` methods), so a path can be thought of as a double-linked list.

The use of the *previous gate* and *next gate* pointers with various gate types is illustrated on figure 4.2.

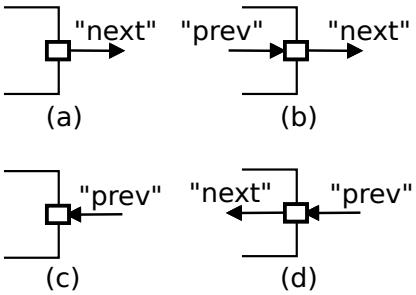


Figure 4.2: (a) simple module output gate, (b) compound module output gate, (c) simple module input gate, (d) compound module input gate

The start and end gates of the path can be found with the `getPathStartGate()` and `getPathEndGate()` methods, which simply follow the *previous gate* and *next gate* pointers, respectively, until they are `nullptr`.

The `isConnectedOutside()` and `isConnectedInside()` methods return whether a gate is connected on the outside or on the inside. They examine either the *previous* or the *next* pointer, depending on the gate type (input or output). For example, an output gate is *connected outside* if the *next* pointer is non-`nullptr`; the same function for an input gate checks the *previous* pointer. Again, see figure 4.2 for an illustration.

The `isConnected()` method is a bit different: it returns true if the gate is *fully* connected, that is, for a compound module gate both inside and outside, and for a simple module gate, outside.

The following code prints the name of the gate a simple module gate is connected to:

```
cGate *gate = gate("somegate");
cGate *otherGate = gate->getType()==cGate::OUTPUT ? gate->getNextGate() :
                                                    gate->getPreviousGate();
if (otherGate)
  EV << "gate is connected to: " << otherGate->getFullPath() << endl;
else
  EV << "gate not connected" << endl;
```

4.6.3 The Connection's Channel

The channel object associated with a connection is accessible by a pointer stored at the source gate of the connection. The pointer is returned by the `getChannel()` method of the gate:

```
cChannel *channel = gate->getChannel();
```

The result may be `nullptr`, that is, a connection may not have an associated channel object.

If you have a channel pointer, you can get back its source gate with the `getSourceGate()` method:

```
cGate *gate = channel->getSourceGate();
```

`cChannel` is just an abstract base class for channels, so to access details of the channel you might need to cast the resulting pointer into a specific channel class, for example `cDelayChannel` or `cDatarateChannel`.

Another specific channel type is `cIdealChannel`, which basically does nothing: it acts as if there was no channel object assigned to the connection. OMNEST sometimes transparently inserts a `cIdealChannel` into a channel-less connection, for example to hold the display string associated with the connection.

Often you are not really interested in a specific connection's channel, but rather in the *transmission channel* (see 4.7.6) of the connection path that starts at a specific output gate. The transmission channel can be found by following the connection path until you find a channel whose `isTransmissionChannel()` method returns `true`, but `cGate` has a convenience method for this, named `getTransmissionChannel()`. An example usage:

```
cChannel *txChan = gate("ppp$o")->getTransmissionChannel();
```

A complementer method to `getTransmissionChannel()` is `getIncomingTransmissionChannel()`; it is usually invoked on input gates, and searches the connection path in reverse direction.

```
cChannel *incomingTxChan = gate("ppp$i")->getIncomingTransmissionChannel();
```

Both methods throw an error if no transmission channel is found. If this is not suitable, use the similar `findTransmissionChannel()` and `findIncomingTransmissionChannel()` methods that simply return `nullptr` in that case.

Channels are covered in more detail in section 4.8.

4.7 Sending and Receiving Messages

On an abstract level, an OMNEST simulation model is a set of simple modules that communicate with each other via message passing. The essence of simple modules is that they create, send, receive, store, modify, schedule and destroy messages – the rest of OMNEST exists to facilitate this task, and collect statistics about what was going on.

Messages in OMNEST are instances of the `cMessage` class or one of its subclasses. Network packets are represented with `cPacket`, which is also subclassed from `cMessage`. Message objects are created using the C++ `new` operator, and destroyed using the `delete` operator when they are no longer needed.

Messages are described in detail in chapter 5. At this point, all we need to know about them is that they are referred to as `cMessage *` pointers. In the examples below, messages will be created with `new cMessage("foo")` where `"foo"` is a descriptive message name, used for visualization and debugging purposes.

4.7.1 Self-Messages

Nearly all simulation models need to schedule future events in order to implement timers, timeouts, delays, etc. Some typical examples:

- A source module that periodically creates and sends messages needs to schedule the next send after every send operation;
- A server which processes jobs from a queue needs to start a timer every time it begins processing a job. When the timer expires, the finished job can be sent out, and a new job may start processing;

- When a packet is sent by a communications protocol that employs retransmission, it needs to schedule a timeout so that the packet can be retransmitted if no acknowledge arrives within a certain amount of time.

In OMNEST, you solve such tasks by letting the simple module send a message to itself; the message would be delivered to the simple module at a later point of time. Messages used this way are called *self-messages*, and the module class has special methods for them that allow for implementing self-messages without gates and connections.

Scheduling an Event

The module can send a message to itself using the `scheduleAt()` function. `scheduleAt()` accepts an *absolute* simulation time:

```
scheduleAt(t, msg);
```

Since the target time is often relative to the current simulation time, the function has another variant, `scheduleAfter()`, which takes a *delta* instead of an absolute simulation time. The following calls are equivalent:

```
scheduleAt(simTime()+delta, msg);
scheduleAfter(delta, msg);
```

Self-messages are delivered to the module in the same way as other messages (via the usual receive calls or `handleMessage()`); the module may call the `isSelfMessage()` member of any received message to determine if it is a self-message.

You can determine whether a message is currently in the FES by calling its `isScheduled()` member function.

Cancelling an Event

Scheduled self-messages can be cancelled (i.e. removed from the FES). This feature facilitates implementing timeouts.

```
cancelEvent(msg);
```

The `cancelEvent()` function takes a pointer to the message to be cancelled, and also returns the same pointer. After having it cancelled, you may delete the message or reuse it in subsequent `scheduleAt()` calls. `cancelEvent()` has no effect if the message is not scheduled at that time.

There is also a convenience method called `cancelAndDelete()` implemented as `if (msg!=nullptr) delete cancelEvent(msg)`; this method is primarily useful for writing destructors.

The following example shows how to implement a timeout in a simple imaginary stop-and-wait protocol. The code utilizes a `timeoutEvent` module class data member that stores the pointer of the `cMessage` used as self-message, and compares it to the pointer of the received message to identify whether a timeout has occurred.

```
void Protocol::handleMessage(cMessage *msg)
{
    if (msg == timeoutEvent) {
        // timeout expired, re-send packet and restart timer
        send(currentPacket->dup(), "out");
```

```
            scheduleAt(simTime() + timeout, timeoutEvent);
        }
        else if (...) {  // if acknowledgement received
            // cancel timeout, prepare to send next packet, etc.
            cancelEvent(timeoutEvent);
            ...
        }
        else {
            ...
        }
    }
```

Re-scheduling an Event

To reschedule an event which is currently scheduled to a different simulation time, it first needs to be cancelled using `cancelEvent()`. This is shown in the following example code:

```
if (msg->isScheduled())
    cancelEvent(msg);
scheduleAt(simTime() + delay, msg);
```

For convenience, the above functionality is available as a single call, as the functions `rescheduleAt()` and `rescheduleAfter()`. The first one takes an absolute simulation time, the second one a *delta* relative to the current simulation time.

```
rescheduleAt(t, msg);
rescheduleAfter(delta, msg);
```

Using these dedicated functions may be more efficient than the `cancelEvent()`+`scheduleAt()` combo.

4.7.2 Sending Messages

Once created, a message object can be sent through an output gate using one of the following functions:

```
send(cMessage *msg, const char *gateName, int index=0);
send(cMessage *msg, int gateId);
send(cMessage *msg, cGate *gate);
```

In the first function, the argument `gateName` is the name of the gate the message has to be sent through. If this gate is a vector gate, `index` determines though which particular output gate this has to be done; otherwise, the `index` argument is not needed.

The second and third functions use the gate ID and the pointer to the gate object. They are faster than the first one because they don't have to search for the gate by name.

Examples:

```
send(msg, "out");
send(msg, "outv", i); // send via a gate in a gate vector
```

To send via an inout gate, remember that an inout gate is an input and an output gate glued together, and the two halves can be identified with the `$i` and `$o` name suffixes. Thus, the gate name needs to be specified in the `send()` call with the `$o` suffix:

```
send(msg, "g$o");
send(msg, "g$o", i); // if "g[]" is a gate vector
```

4.7.3 Broadcasts and Retransmissions

When implementing broadcasts or retransmissions, two frequently occurring tasks in protocol simulation, you might feel tempted to use the same message in multiple `send()` operations. Do not do it – you cannot send the same message object multiple times. Instead, duplicate the message object.

Why? A message is like a real-world object – it cannot be at two places at the same time. Once sent out, the message no longer belongs to the module: it is taken over by the simulation kernel, and will eventually be delivered to the destination module. The sender module should not even refer to its pointer any more. Once the message arrives in the destination module, that module will have full authority over it – it can send it on, destroy it immediately, or store it for further handling. The same applies to messages that have been scheduled – they belong to the simulation kernel until they are delivered back to the module.

To enforce the rules above, all message sending functions check that the module actually owns the message it is about to send. If the message is in another module, in a queue, currently scheduled, etc., a runtime error will be generated: *not owner of message*. [6]

Broadcasting Messages

In your model, you may need to broadcast a message to several destinations. Broadcast can be implemented in a simple module by sending out copies of the same message, for example on every gate of a gate vector. As described above, you cannot use the same message pointer for in all `send()` calls – what you have to do instead is create copies (duplicates) of the message object and send them.

Example:

```
for (int i = 0; i < n; i++)
{
    cMessage *copy = msg->dup();
    send(copy, "out", i);
}
delete msg;
```

You might have noticed that copying the message for the last gate is redundant: we can just send out the original message there. Also, we can utilize gate IDs to avoid looking up the gate by name for each send operation. We can exploit the fact that the ID of gate k in a gate vector can be produced as $baseID + k$. The optimized version of the code looks like this:

```
int outGateBaseId = gateBaseId("out");
for (int i = 0; i < n; i++)
    send(i==n-1 ? msg : msg->dup(), outGateBaseId+i);
```

[6] The feature does not increase runtime overhead significantly, because it uses the object ownership management (described in Section 7.14); it merely checks that the owner of the message is the module that wants to send it.

Retransmissions

Many communication protocols involve retransmissions of packets (frames). When implementing retransmissions, you cannot just hold a pointer to the same message object and send it again and again – you'd get the *not owner of message* error on the first resend.

Instead, for (re)transmission, you should create and send copies of the message, and retain the original. When you are sure there will not be any more retransmission, you can delete the original message.

Creating and sending a copy:

```
// (re)transmit packet:
cMessage *copy = packet->dup();
send(copy, "out");
```

and finally (when no more retransmissions will occur):

```
delete packet;
```

4.7.4 Delayed Sending

Sometimes it is necessary for module to hold a message for some time interval, and then send it. This can be achieved with self-messages, but there is a more straightforward method: delayed sending. The following methods are provided for delayed sending:

```
sendDelayed(cMessage *msg, double delay, const char *gateName, int index);
sendDelayed(cMessage *msg, double delay, int gateId);
sendDelayed(cMessage *msg, double delay, cGate *gate);
```

The arguments are the same as for `send()`, except for the extra *delay* parameter. The delay value must be non-negative. The effect of the function is similar to as if the module had kept the message for the delay interval and sent it afterwards; even the *sending time* timestamp of the message will be set to the current simulation time plus *delay*.

A example call:

```
sendDelayed(msg, 0.005, "out");
```

The `sendDelayed()` function does not internally perform a `scheduleAt()` followed by a `send()`, but rather it computes everything about the message sending up front, including the arrival time and the target module. This has two consequences. First, `sendDelayed()` is more efficient than a `scheduleAt()` followed by a `send()` because it eliminates one event. The second, less pleasant consequence is that changes in the connection path during the delay will *not* be taken into account (because everything is calculated in advance, before the changes take place).

> **NOTE:** The fact that `sendDelayed()` computes the message arrival information up front does not make a difference if the model is static, but may lead to surprising results if the model changes in time. For example, if a connection in the path gets deleted, disabled, or reconnected to another module during the delay period, the message will still be delivered to the original module as if nothing happened.
> Therefore, despite its performance advantage, you should think twice before using `sendDelayed()` in a simulation model. It may have its place in a one-shot simulation model that you know is static, but it certainly should be avoided in reusable modules that need to work correctly in a wide variety of simulation models.

4.7.5 Direct Message Sending

At times it is covenient to be able to send a message directly to an input gate of another module. The `sendDirect()` function is provided for this purpose.

This function has several flavors. The first set of `sendDirect()` functions accept a message and a target gate; the latter can be specified in various forms:

```
sendDirect(cMessage *msg, cModule *mod, int gateId)
sendDirect(cMessage *msg, cModule *mod, const char *gateName, int index=-1)
sendDirect(cMessage *msg, cGate *gate)
```

An example for direct sending:

```
cModule *targetModule = getParentModule()->getSubmodule("node2");
sendDirect(new cMessage("msg"), targetModule, "in");
```

At the target module, there is no difference between messages received directly and those received over connections.

The target gate must be an unconnected gate; in other words, modules must have dedicated gates to be able to receive messages sent via `sendDirect()`. You cannot have a gate which receives messages via both connections and `sendDirect()`.

It is recommended to tag gates dedicated for receiving messages via `sendDirect()` with the `@directIn` property in the module's NED declaration. This will cause OMNEST not to complain that the gate is not connected in the network or compound module where the module is used.

An example:

```
simple Radio {
    gates:
        input radioIn @directIn; // for receiving air frames
}
```

The target module is usually a simple module, but it can also be a compound module. The message will follow the connections that start at the target gate, and will be delivered to the module at the end of the path – just as with normal connections. The path must end in a simple module.

It is even permitted to send to an output gate, which will also cause the message to follow the connections starting at that gate. This can be useful, for example, when several submodules are sending to a single output gate of their parent module.

A second set of `sendDirect()` methods accept a propagation delay and a transmission duration as parameters as well:

```
sendDirect(cMessage *msg, simtime_t propagationDelay, simtime_t duration,
           cModule *mod, int gateId)
sendDirect(cMessage *msg, simtime_t propagationDelay, simtime_t duration,
           cModule *mod, const char *gateName, int index=-1)
sendDirect(cMessage *msg, simtime_t propagationDelay, simtime_t duration,
           cGate *gate)
```

The transmission duration parameter is important when the message is also a packet (instance of `cPacket`). For messages that are not packets (not subclassed from `cPacket`), the duration parameter is ignored.

If the message is a packet, the duration will be written into the packet, and can be read by the receiver with the `getDuration()` method of the packet.

The receiver module can choose whether it wants the simulation kernel to deliver the packet object to it at the start or at the end of the reception. The default is the latter; the module can change it by calling `setDeliverImmediately()` on the final input gate, that is, on `targetGate->getPathEndGate()`.

4.7.6 Packet Transmissions

When a message is sent out on a gate, it usually travels through a series of connections until it arrives at the destination module. We call this series of connections a *connection path*.

Several connections in the path may have an associated channel, but there can be only one channel per path that models nonzero transmission duration. This restriction is enforced by the simulation kernel. This channel is called the *transmission channel*. [7]

> **NOTE:** In practice, this means that there can be only one `ned.DatarateChannel` in the path. Note that unnamed channels with a `datarate` parameter also map to `ned.DatarateChannel`.

Transmitting a Packet

Packets may only be sent when the transmission channel is idle. This means that after each transmission, the sender module needs to wait until the channel has finished transmitting before it can send another packet.

You can get a pointer to the transmission channel by calling the `getTransmissionChannel()` method on the output gate. The channel's `isBusy()` and `getTransmissionFinishTime()` methods can tell you whether a channel is currently transmitting, and when the transmission is going to finish. (When the latter is less or equal the current simulation time, the channel is free.) If the channel is currently busy, sending needs to be postponed: the packet can be stored in a queue, and a timer (self-message) can be scheduled for the time when the channel becomes empty.

A code example to illustrate the above process:

```
cPacket *pkt = ...; // packet to be transmitted
cChannel *txChannel = gate("out")->getTransmissionChannel();
simtime_t txFinishTime = txChannel->getTransmissionFinishTime();
if (txFinishTime <= simTime()) {
    // channel free; send out packet immediately
    send(pkt, "out");
}
else {
    // store packet and schedule timer; when the timer expires,
    // the packet should be removed from the queue and sent out
    txQueue.insert(pkt);
    scheduleAt(txFinishTime, endTxMsg);
}
```

[7] Moreover, if `sendDirect()` with a nonzero duration was used to send the packet to the start gate of the path, then the path cannot have a transmission channel at all. The point is that the a transission duration must be unambiguous.

NOTE: If there is a channel with a propagation delay in the path before the transmission channel, the delay should be manually substracted from the value returned by `getTransmissionFinishTime()`! The same applies to `isBusy()`: it tells whether the channel is currently busy, and not whether it will be busy when a packet that you send gets there. It is therefore advisable that you never use propagation delays in front of a transmission channel in a path.

The `getTransmissionChannel()` method searches the connection path each time it is called. If performance is important, it is a good idea to obtain the transmission channel pointer once, and then cache it. When the network topology changes, the cached channel pointer needs to be updated; section 4.14.3 describes the mechanism that can be used to get notifications about topology changes.

Receiving a Packet

As a result of error modeling in the channel, the packet may arrive with the *bit error* flag set (`hasBitError() method`. It is the receiver module's responsibility to examine this flag and take appropriate action (i.e. discard the packet).

Normally the packet object gets delivered to the destination module at the simulation time that corresponds to finishing the reception of the message (ie. the arrival of its last bit). However, the receiver module may change this by "reprogramming" the receiver gate with the `setDeliverImmediately()` method:

```
gate("in")->setDeliverImmediately(true);
```

This method may only be called on simple module input gates, and it instructs the simulation kernel to deliver packets arriving through that gate at the simulation time that corresponds to the beginning of the reception process. `getDeliverOnReceptionStart()` only needs to be called once, so it is usually done in the `initialize()` method of the module.

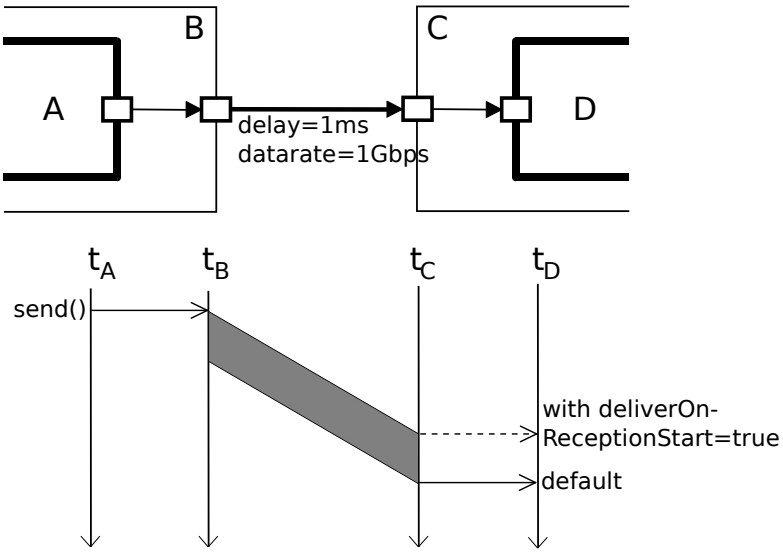


Figure 4.3: Packet transmission

When a packet is delivered to the module, the packet's `isReceptionStart()` method can be called to determine whether it corresponds to the start or end of the reception process (it should be the same as the `getDeliverOnReceptionStart()` flag of the input gate), and `getDuration()` returns the transmission duration.

The following example code prints the start and end times of a packet reception:

```
simtime_t startTime, endTime;
if (pkt->isReceptionStart()) {
    // gate was reprogrammed with setDeliverImmediately(true)
    startTime = pkt->getArrivalTime(); // or: simTime();
    endTime = startTime + pkt->getDuration();
}
else {
    // default case
    endTime = pkt->getArrivalTime(); // or: simTime();
    startTime = endTime - pkt->getDuration();
}
EV << "interval: " << startTime << ".." << endTime << "\n";
```

Note that this works with wireless connections (`sendDirect()`) as well; there, the duration is an argument to the `sendDirect()` call.

Aborting Transmissions

Certain protocols, for example Ethernet require the ability to abort a transmission before it completes. The support OMNEST provides for this task is the `forceTransmissionFinishTime()` channel method. This method forcibly overwrites the *transmissionFinishTime* member of the channel with the given value, allowing the sender to transmit another packet without raising the *"channel is currently busy"* error. The receiving party needs to be notified about the aborted transmission by some external means, for example by sending another packet or an out-of-band message.

Implementation of Message Sending

Message sending is implemented like this: the arrival time and the bit error flag of a message are calculated right inside the `send()` call, then the message is inserted into the FES with the calculated arrival time. The message does *not* get scheduled individually for each link. This implementation was chosen because of its run-time efficiency.

> **NOTE:** The consequence of this implementation is that any change in the channel's parameters (delay, data rate, bit error rate, etc.) will only affect messages *sent* after the change. Messages already underway will not be influenced by the change.
> This is not a huge problem in practice, but if it is important to model channels with changing parameters, the solution is to insert simple modules into the path to ensure strict scheduling.

The code which inserts the message into the FES is the `arrived()` method of the recipient module. By overriding this method it is possible to perform custom processing at the recipient module immediately, still from within the `send()` call. Use only if you know what you are doing!

4.7.7 Receiving Messages with activity()

Receiving Messages

`activity()`-based modules receive messages with the `receive()` method of `cSimpleModule`. `receive()` cannot be used with `handleMessage()`-based modules.

```
cMessage *msg = receive();
```

The `receive()` function accepts an optional *timeout* parameter. (This is a *delta*, not an absolute simulation time.) If no message arrives within the timeout period, the function returns `nullptr`. [8]

```
simtime_t timeout = 3.0;
cMessage *msg = receive(timeout);

if (msg==nullptr) {
    ...   // handle timeout
}
else {
    ...  // process message
}
```

The wait() Function

The `wait()` function suspends the execution of the module for a given amount of simulation time (a *delta*). `wait()` cannot be used with `handleMessage()`-based modules.

```
wait(delay);
```

In other simulation software, `wait()` is often called *hold*. Internally, the `wait()` function is implemented by a `scheduleAt()` followed by a `receive()`. The `wait()` function is very convenient in modules that do not need to be prepared for arriving messages, for example message generators. An example:

```
for (;;) {
    // wait for some, potentially random, amount of time, specified
    // in the interarrivalTime volatile module parameter
    wait(par("interarrivalTime").doubleValue());

    // generate and send message
    ...
}
```

It is a runtime error if a message arrives during the wait interval. If you expect messages to arrive during the wait period, you can use the `waitAndEnqueue()` function. It takes a pointer to a queue object (of class `cQueue`, described in chapter 7) in addition to the wait interval. Messages that arrive during the wait interval are accumulated in the queue, and they can be processed after the `waitAndEnqueue()` call returns.

```
cQueue queue("queue");
```

[8] Putaside-queue and the functions `receiveOn()`, `receiveNew()`, and `receiveNewOn()` were deprecated in OMNEST 2.3 and removed in OMNEST 3.0.

```
...
waitAndEnqueue(waitTime, &queue);
if (!queue.empty())
{
    // process messages arrived during wait interval
    ...
}
```

4.8 Channels

4.8.1 Overview

Channels encapsulate parameters and behavior associated with connections. Channel types are like simple modules, in the sense that they are declared in NED, and there are C++ implementation classes behind them. Section 3.5 describes NED language support for channels, and explains how to associate C++ classes with channel types declared in NED.

C++ channel classes must subclass from the abstract base class `cChannel`. However, when creating a new channel class, it may be more practical to extend one of the existing C++ channel classes behind the three predefined NED channel types:

- `cIdealChannel` implements the functionality of `ned.IdealChannel`
- `cDelayChannel` implements the functionality of `ned.DelayChannel`
- `cDatarateChannel` implements the functionality of `ned.DatarateChannel`

Channel classes need to be registered with the `Define_Channel()` macro, just like simple module classes need `Define_Module()`.

The channel base class `cChannel` inherits from `cComponent`, so channels participate in the initialization and finalization protocol (`initialize()` and `finish()`) described in 4.3.3.

The parent module of a channel (as returned by the `getParentModule()`) is the module that contains the connection. If a connection connects two modules that are children of the same compound module, the channel's parent is the compound module. If the connection connects a compound module to one of its submodules, the channel's parent is also the compound module.

4.8.2 The Channel API

When subclassing `Channel`, the following pure virtual member functions need to be overridden:

- `bool isTransmissionChannel() const`
- `simtime_t getTransmissionFinishTime() const`
- `void processMessage(cMessage *msg, simtime_t t, result_t& result)`

The first two functions are usually one-liners; the channel behavior is encapsulated in the third function, `processMessage()`.

Transmission Channels

The first function, `isTransmissionChannel()`, determines whether the channel is a *transmission channel*, i.e. one that models transmission duration. A transmission channel sets the duration field of packets sent through it (see the `setDuration()` field of `cPacket`).

The `getTransmissionFinishTime()` function is only used with transmission channels, and it should return the simulation time the sender will finish (or has finished) transmitting. This method is called by modules that send on a transmission channel to find out when the channel becomes available. The channel's `isBusy()` method is implemented simply as `return getTransmissionFinishTime() < simTime()`. For non-transmission channels, the `getTransmissionFinishTime()` return value may be any simulation time which is less than or equal to the current simulation time.

The processMessage() Function

The third function, `processMessage()` encapsulates the channel's functionality. However, before going into the details of this function we need to understand how OMNEST handles message sending on connections.

Inside the `send()` call, OMNEST follows the connection path denoted by the `getNextGate()` functions of gates, until it reaches the target module. At each "hop", the corresponding connection's channel (if the connection has one) gets a chance to add to the message's arrival time (*propagation time modeling*), calculate a *transmission duration*, and to modify the message object in various ways, such as set the bit error flag in it (*bit error modeling*). After processing all hops that way, OMNEST inserts the message object into the Future Events Set (FES, see section 4.1.2), and the `send()` call returns. Then OMNEST continues to process events in increasing timestamp order. The message will be delivered to the target module's `handleMessage()` (or `receive()`) function when it gets to the front of the FES.

A few more details: a channel may instruct OMNEST to delete the message instead of inserting it into the FES; this can be useful to model disabled channels, or to model that the message has been lost altogether. The `getDeliverOnReceptionStart()` flag of the final gate in the path will determine whether the transmission duration will be added to the arrival time or not. Packet transmissions have been described in section 4.7.6.

Now, back to the `processMessage()` method.

The method gets called as part of the above process, when the message is processed at the given hop. The method's arguments are the message object, the simulation time the beginning of the message will reach the channel (i.e. the sum of all previous propagation delays), and a struct in which the method can return the results.

The `result_t` struct is an inner type of `cChannel`, and looks like this:

```
struct result_t {
    simtime_t delay;      // propagation delay
    simtime_t duration;   // transmission duration
    bool discard;         // whether the channel has lost the message
};
```

It also has a constructor that initializes all fields to zero; it is left out for brevity.

The method should model the transmission of the given message starting at the given t time, and store the results (propagation delay, transmission duration, deletion flag) in the result object. Only the relevant fields in the result object need to be changed, others can be left

untouched.

Transmission duration and bit error modeling only applies to packets (i.e. to instances of `cPacket`, where `cMessage`'s `isPacket()` returns true); it should be skipped for non-packet messages. `processMessage()` does not need to call the `setDuration()` method on the packet; this is done by the simulation kernel. However, it should call `setBitError(true)` on the packet if error modeling results in bit errors.

If the method sets the `discard` flag in the result object, that means that the message object will be deleted by OMNEST; this facility can be used to model that the message gets lost in the channel.

The `processMessage()` method does not need to throw error on overlapping transmissions, or if the packet's duration field is already set; these checks are done by the simulation kernel before `processMessage()` is called.

4.8.3 Channel Examples

To illustrate coding channel behavior, we look at how the built-in channel types are implemented.

`cIdealChannel` lets through messages and packets without any delay or change. Its `isTransmissionChannel()` method returns `false`, `getTransmissionFinishTime()` returns $0s$, and the body of its `processMessage()` method is empty:

```
void cIdealChannel::processMessage(cMessage *msg, simtime_t t, result_t& result)
{
}
```

`cDelayChannel` implements propagation delay, and it can be disabled; in its disabled state, messages sent though it will be discarded. This class still models zero transmission duration, so its `isTransmissionChannel()` and `getTransmissionFinishTime()` methods still return `false` and $0s$. The `processMessage()` method sets the appropriate fields in the `result_t` struct:

```
void cDelayChannel::processMessage(cMessage *msg, simtime_t t, result_t& result)
{
    // if channel is disabled, signal that message should be deleted
    result.discard = isDisabled;

    // propagation delay modeling
    result.delay = delay;
}
```

The `handleParameterChange()` method is also redefined, so that the channel can update its internal `delay` and `isDisabled` data members if the corresponding channel parameters change during simulation. [9]

`cDatarateChannel` is different. It performs model packet duration (duration is calculated from the data rate and the length of the packet), so `isTransmissionChannel()` returns `true`. `getTransmissionFinishTime()` returns the value of a `txfinishtime` data member, which gets updated after every packet.

```
simtime_t cDatarateChannel::getTransmissionFinishTime() const
```

[9] This code is a little simplified; the actual code uses a bit in a bitfield to store the value of `isDisabled`.

```
{
    return txfinishtime;
}
```

cDatarateChannel's processMessage() method makes use of the isDisabled, datarate, ber and per data members, which are also kept up to date with the help of handleParameterChange().

```
void cDatarateChannel::processMessage(cMessage *msg, simtime_t t, result_t& result
{
    // if channel is disabled, signal that message should be deleted
    if (isDisabled) {
        result.discard = true;
        return;
    }

    // datarate modeling
    if (datarate!=0 && msg->isPacket()) {
        simtime_t duration = ((cPacket *)msg)->getBitLength() / datarate;
        result.duration = duration;
        txfinishtime = t + duration;
    }
    else {
        txfinishtime = t;
    }

    // propagation delay modeling
    result.delay = delay;

    // bit error modeling
    if ((ber!=0 || per!=0) && msg->isPacket()) {
        cPacket *pkt = (cPacket *)msg;
        if (ber!=0 && dblrand() < 1.0 - pow(1.0-ber, (double)pkt->getBitLength())
            pkt->setBitError(true);
        if (per!=0 && dblrand() < per)
            pkt->setBitError(true);
    }
}
```

4.9 Stopping the Simulation

4.9.1 Normal Termination

You can finish the simulation with the endSimulation() function:

```
endSimulation();
```

endSimulation() is rarely needed in practice because you can specify simulation time and CPU time limits in the ini file (see later).

4.9.2 Raising Errors

When the simulation encounters an error condition, it can throw a `cRuntimeError` exception to terminate the simulation with an error message. (Under Cmdenv, the exception also causes a nonzero program exit code). The `cRuntimeError` class has a constructor with a `printf()`-like argument list. An example:

```
if (windowSize <= 0)
    throw cRuntimeError("Invalid window size %d; must be >=1", windowSize);
```

Do not include newline (`\n`), period or exclamation mark in the error text; it will be added by OMNEST.

The same effect can be achieved by calling the `error()` method of `cModule`:

```
if (windowSize <= 0)
    error("Invalid window size %d; must be >=1", windowSize);
```

Of course, the `error()` method can only be used when a module pointer is available.

4.10 Finite State Machines

4.10.1 Overview

Finite State Machines (FSMs) can make life with `handleMessage()` easier. OMNEST provides a class and a set of macros to build FSMs.

The key points are:

- There are two kinds of states: *transient* and *steady*. On each event (that is, at each call to `handleMessage()`), the FSM transitions out of the current (*steady*) state, undergoes a series of state changes (runs through a number of *transient* states), and finally arrives at another *steady* state. Thus between two events, the system is always in one of the steady states. Transient states are therefore not really a must – they exist only to group actions to be taken during a transition in a convenient way.
- You can assign program code to handle entering and leaving a state (known as entry/exit code). Staying in the same state is handled as leaving and re-entering the state.
- Entry code should not modify the state (this is verified by OMNEST). State changes (transitions) must be put into the exit code.

OMNEST's FSMs *can* be nested. This means that any state (or rather, its entry or exit code) may contain a further full-fledged `FSM_Switch()` (see below). This allows you to introduce sub-states and thereby bring some structure into the state space if it becomes too large.

The FSM API

FSM state is stored in an object of type `cFSM`. The possible states are defined by an enum; the enum is also a place to define which state is transient and which is steady. In the following example, SLEEP and ACTIVE are steady states and SEND is transient (the numbers in parentheses must be unique within the state type and they are used for constructing the numeric IDs for the states):

```
enum {
  INIT = 0,
  SLEEP = FSM_Steady(1),
  ACTIVE = FSM_Steady(2),
  SEND = FSM_Transient(1),
};
```

The actual FSM is embedded in a switch-like statement, `FSM_Switch()`, with cases for entering and leaving each state:

```
FSM_Switch(fsm)
{
  case FSM_Exit(state1):
    //...
    break;
  case FSM_Enter(state1):
    //...
    break;
  case FSM_Exit(state2):
    //...
    break;
  case FSM_Enter(state2):
    //...
    break;
  //...
};
```

State transitions are done via calls to `FSM_Goto()`, which simply stores the new state in the `cFSM` object:

```
FSM_Goto(fsm, newState);
```

The FSM starts from the state with the numeric code 0; this state is conventionally named INIT.

Debugging FSMs

FSMs can log their state transitions, with the output looking like this:

```
...
FSM GenState: leaving state SLEEP
FSM GenState: entering state ACTIVE
...
FSM GenState: leaving state ACTIVE
FSM GenState: entering state SEND
FSM GenState: leaving state SEND
FSM GenState: entering state ACTIVE
...
FSM GenState: leaving state ACTIVE
FSM GenState: entering state SLEEP
...
```

To enable the above output, define `FSM_DEBUG` before including `omnetpp.h`.

```
#define FSM_DEBUG    // enables debug output from FSMs
#include <omnetpp.h>
```

FSMs perform their logging via the `FSM_Print()` macro, defined as something like this:

```
#define FSM_Print(fsm,exiting)
  (EV << "FSM " << (fsm).getName()
      << ((exiting) ? ": leaving state " : ": entering state ")
      << (fsm).getStateName() << endl)
```

The log output format can be changed by undefining `FSM_Print()` after the inclusion of `omnetpp.ini`, and providing a new definition.

Implementation

`FSM_Switch()` is a macro. It expands to a `switch` statement embedded in a `for()` loop which repeats until the FSM reaches a steady state.

Infinite loops are avoided by counting state transitions: if an FSM goes through 64 transitions without reaching a steady state, the simulation will terminate with an error message.

An Example

Let us write another bursty packet generator. It will have two states, SLEEP and ACTIVE. In the SLEEP state, the module does nothing. In the ACTIVE state, it sends messages with a given inter-arrival time. The code was taken from the Fifo2 sample simulation.

```
#define FSM_DEBUG
#include <omnetpp.h>
using namespace omnetpp;

class BurstyGenerator : public cSimpleModule
{
  protected:
    // parameters
    double sleepTimeMean;
    double burstTimeMean;
    double sendIATime;
    cPar *msgLength;

    // FSM and its states
    cFSM fsm;
    enum {
      INIT = 0,
      SLEEP = FSM_Steady(1),
      ACTIVE = FSM_Steady(2),
      SEND = FSM_Transient(1),
    };

    // variables used
    int i;
    cMessage *startStopBurst;
    cMessage *sendMessage;
```

```
    // the virtual functions
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(BurstyGenerator);

void BurstyGenerator::initialize()
{
    fsm.setName("fsm");
    sleepTimeMean = par("sleepTimeMean");
    burstTimeMean = par("burstTimeMean");
    sendIATime = par("sendIATime");
    msgLength = &par("msgLength");
    i = 0;
    WATCH(i); // always put watches in initialize()
    startStopBurst = new cMessage("startStopBurst");
    sendMessage = new cMessage("sendMessage");
    scheduleAt(0.0,startStopBurst);
}

void BurstyGenerator::handleMessage(cMessage *msg)
{
   FSM_Switch(fsm) {
     case FSM_Exit(INIT):
       // transition to SLEEP state
       FSM_Goto(fsm,SLEEP);
       break;
     case FSM_Enter(SLEEP):
       // schedule end of sleep period (start of next burst)
       scheduleAt(simTime()+exponential(sleepTimeMean), startStopBurst);
     break;
     case FSM_Exit(SLEEP):
       // schedule end of this burst
       scheduleAt(simTime()+exponential(burstTimeMean), startStopBurst);
       // transition to ACTIVE state:
       if (msg!=startStopBurst) {
         error("invalid event in state ACTIVE");
       }
       FSM_Goto(fsm,ACTIVE);
       break;
     case FSM_Enter(ACTIVE):
       // schedule next sending
       scheduleAt(simTime()+exponential(sendIATime), sendMessage);
     break;
     case FSM_Exit(ACTIVE):
       // transition to either SEND or SLEEP
       if (msg==sendMessage) {
         FSM_Goto(fsm,SEND);
       } else if (msg==startStopBurst) {
         cancelEvent(sendMessage);
```

```
          FSM_Goto(fsm,SLEEP);
        } else {
          error("invalid event in state ACTIVE");
        }
        break;
      case FSM_Exit(SEND): {
        // generate and send out job
        char msgname[32];
        sprintf(msgname, "job-%d", ++i);
        EV << "Generating " << msgname << endl;
        cMessage *job = new cMessage(msgname);
        job->setBitLength((long) *msgLength);
        job->setTimestamp();
        send(job, "out");
        // return to ACTIVE
        FSM_Goto(fsm,ACTIVE);
        break;
      }
    }
}
```

4.11 Navigating the Module Hierarchy

4.11.1 Module Vectors

If a module is part of a module vector, the `getIndex()` and `getVectorSize()` member functions can be used to query its index and the vector size:

```
EV << "This is module [" << module->getIndex() <<
      "] in a vector of size [" << module->getVectorSize() << "].\n";
```

4.11.2 Component IDs

Every component (module and channel) in the network has an ID that can be obtained from `cComponent`'s `getId()` member function:

```
int componentId = getId();
```

IDs uniquely identify a module or channel for the whole duration of the simulation. This holds even when modules are created and destroyed dynamically, because IDs of deleted modules or channels are never reused for newly created ones.

To look up a component by ID, one needs to use methods of the simulation manager object, `cSimulation`. `getComponent()` expects an ID, and returns the component's pointer if the component still exists, otherwise it returns `nullptr`. The method has two variations, `getModule(id)` and `getChannel(id)`. They return `cModule` and `cChannel` pointers if the identified component is in fact a module or a channel, respectively, otherwise they return `nullptr`.

```
int id = 100;
cModule *mod = getSimulation()->getModule(id); // exists, and is a module
```

4.11.3 Walking Up and Down the Module Hierarchy

The parent module can be accessed by the `getParentModule()` member function:

```
cModule *parent = getParentModule();
```

For example, the parameters of the parent module are accessed like this:

```
double timeout = getParentModule()->par("timeout");
```

`cModule`'s `findSubmodule()` and `getSubmodule()` member functions make it possible to look up the module's submodules by name (or name and index if the submodule is in a module vector). The first one returns the module ID of the submodule, and the latter returns the module pointer. If the submodule is not found, they return -1 or `nullptr`, respectively.

```
int submodID = module->findSubmodule("foo", 3); // look up "foo[3]"
cModule *submod = module->getSubmodule("foo", 3);
```

4.11.4 Finding Modules by Path

`cModule`'s `getModuleByPath()` member function can be used to find modules by relative or absolute path. It accepts a path string, and returns the pointer of the matching module, or throws an exception if it was not found. If it is not known in advance whether the module exists, its companion function `findModuleByPath()` can be used. `findModuleByPath()` returns `nullptr` if the module identified by the path does not exist, but otherwise behaves identically to `getModuleByPath()`.[10]

The path is dot-separated list of module names. The special module name ^ (caret) stands for the parent module. If the path starts with a dot or caret, it is understood as relative to this module, otherwise it is taken to mean an absolute path. For absolute paths, inclusion of the toplevel module's name in the path is optional. The toplevel module itself may be referred to as `<root>`.

The following lines demonstrate relative paths, and find the `app[3]` submodule and the `gen` submodule of the `app[3]` submodule of the module in question:

```
cModule *app = module->getModuleByPath(".app[3]");  // note leading dot
cModule *gen = module->getModuleByPath(".app[3].gen");
```

Without the leading dot, the path is interpreted as absolute. The following lines both find the `tcp` submodule of `host[2]` in the network, regardless of the module on which the `getModuleByPath()` has been invoked.

```
cModule *tcp = module->getModuleByPath("Network.host[2].tcp");
cModule *tcp = module->getModuleByPath("host[2].tcp");
```

The parent module may be expressed with a caret:

```
cModule *parent = module->getModuleByPath("^"); // parent module
cModule *tcp = module->getModuleByPath("^.tcp"); // sibling module
cModule *other = module->getModuleByPath("^.^.host[1].tcp"); // two levels up, the
```

[10] `findModuleByPath()` was introduced in OMNeT++ 6.0. In previous versions, `getModuleByPath()` returned `nullptr` if there was no matching module.

4.11.5 Iterating over Submodules

To access all modules within a compound module, one can use `cModule::SubmoduleIterator`.

```
for (cModule::SubmoduleIterator it(module); !it.end(); it++) {
    cModule *submodule = *it;
    EV << submodule->getFullName() << endl;
}
```

4.11.6 Navigating Connections

To determine the module at the other end of a connection, use `cGate`'s `getPreviousGate()`, `getNextGate()` and `getOwnerModule()` methods. An example:

```
cModule *neighbour = gate("out")->getNextGate()->getOwnerModule();
```

For input gates, use `getPreviousGate()` instead of `getNextGate()`.

The endpoints of the connection path are returned by the `getPathStartGate()` and `getPathEndGate()` `cGate` methods. These methods follow the connection path by repeatedly calling `getPreviousGate()` and `getNextGate()`, respectively, until they arrive at a `nullptr`. An example:

```
cModule *peer = gate("out")->getPathEndGate()->getOwnerModule();
```

4.12 Direct Method Calls Between Modules

In some simulation models, there might be modules which are too tightly coupled for message-based communication to be efficient. In such cases, the solution might be calling one simple module's public C++ methods from another module.

Simple modules are C++ classes, so normal C++ method calls will work. Two issues need to be mentioned, however:

- how to get a pointer to the object representing the module;
- how to let the simulation kernel know that a method call across modules is taking place.

Typically, the called module is in the same compound module as the caller, so the `getParentModule()` and `getSubmodule()` methods of `cModule` can be used to get a `cModule*` pointer to the called module. (Further ways to obtain the pointer are described in the section 4.11.) The `cModule*` pointer then has to be cast to the actual C++ class of the module, so that its methods become visible.

This makes the following code:

```
cModule *targetModule = getParentModule()->getSubmodule("foo");
Foo *target = check_and_cast<Foo *>(targetModule);
target->doSomething();
```

The `check_and_cast<>()` template function on the second line is part of OMNEST. It performs a standard C++ `dynamic_cast`, and checks the result: if it is `nullptr`, `check_and_cast` raises an OMNEST error. Using `check_and_cast` saves you from writing error checking

code: if `targetModule` from the first line is `nullptr` because the submodule named `"foo"` was not found, or if that module is actually not of type `Foo`, an exception is thrown from `check_and_cast` with an appropriate error message.[11]

The second issue is how to let the simulation kernel know that a method call across modules is taking place. Why is this necessary in the first place? First, the simulation kernel always has to know which module's code is currently executing, in order for ownership handling and other internal mechanisms to work correctly. Second, the Qtenv simulation GUI can animate method calls, but to be able to do that, it needs to know about them. Third, method calls are also recorded in the event log.

The solution is to add the `Enter_Method()` or `Enter_Method_Silent()` macro at the top of the methods that may be invoked from other modules. These calls perform context switching, and, in case of `Enter_Method()`, notify the simulation GUI so that animation of the method call can take place. `Enter_Method_Silent()` does not animate the method call, but otherwise it is equivalent `Enter_Method()`. Both macros accept a `printf()`-like argument list (it is optional for `Enter_Method_Silent()`), which should produce a string with the method name and the actual arguments as much as practical. The string is displayed in the animation (`Enter_Method()` only) and recorded into the event log.

```
void Foo::doSomething()
{
    Enter_Method("doSomething()");
    ...
}
```

4.13 Dynamic Module Creation

4.13.1 When To Use

Certain simulation scenarios require the ability to dynamically create and destroy modules. For example, simulating the arrival and departure of new users in a mobile network may be implemented in terms of adding and removing modules during the course of the simulation. Loading and instantiating network topology (i.e. nodes and links) from a data file is another common technique enabled by dynamic module (and link) creation.

OMNEST allows both simple and compound modules to be created at runtime. When instantiating a compound module, its full internal structure (submodules and internal connections) is reproduced.

Once created and started, dynamic modules aren't any different from "static" modules.

4.13.2 Overview

To understand how dynamic module creation works, you have to know a bit about how OMNEST normally instantiates modules. Each module type (class) has a corresponding factory object of the class `cModuleType`. This object is created under the hood by the `Define_Module()` macro, and it has a factory method which can instantiate the module class (this function basically only consists of a `return new <moduleclass>(...)` statement).

[11] A `check_and_cast_nullable<>()` function also exists. It accepts `nullptr` as input, and only complains if the cast goes wrong.

The cModuleType object can be looked up by its name string (which is the same as the module class name). Once you have its pointer, it is possible to call its factory method and create an instance of the corresponding module class – without having to include the C++ header file containing module's class declaration into your source file.

The cModuleType object also knows what gates and parameters the given module type has to have. (This info comes from NED files.)

Simple modules can be created in one step. For a compound module, the situation is more complicated, because its internal structure (submodules, connections) may depend on parameter values and gate vector sizes. Thus, for compound modules it is generally required to first create the module itself, second, set parameter values and gate vector sizes, and then call the method that creates its submodules and internal connections.

As you know already, simple modules with activity() need a starter message. For statically created modules, this message is created automatically by OMNEST, but for dynamically created modules, you have to do this explicitly by calling the appropriate functions.

Calling initialize() has to take place after insertion of the starter messages, because the initializing code may insert new messages into the FES, and these messages should be processed *after* the starter message.

4.13.3 Creating Modules

The first step is to find the factory object. The cModuleType::get() function expects a fully qualified NED type name, and returns the factory object:

```
cModuleType *moduleType = cModuleType::get("foo.nodes.WirelessNode");
```

The return value does not need to be checked for nullptr, because the function raises an error if the requested NED type is not found. (If this behavior is not what you need, you can use the similar cModuleType::find() function, which returns nullptr if the type was not found.)

The All-in-One Method

cModuleType has a createScheduleInit(const char *name, cModule *parentmod) convenience function to get a module up and running in one step.

```
cModule *mod = moduleType->createScheduleInit("node", this);
```

createScheduleInit() performs the following steps: create(), finalizeParameters(), buildInside(), scheduleStart(now) and callInitialize().

This method can be used for both simple and compound modules. Its applicability is somewhat limited, however: because it does everything in one step, you do not have the chance to set parameters or gate sizes, and to connect gates before initialize() is called. (initialize() expects all parameters and gates to be in place and the network fully built when it is called.) Because of the above limitation, this function is mainly useful for creating basic simple modules.

The Detailed Procedure

If the createScheduleInit() all-in-one method is not applicable, one needs to use the full procedure. It consists of five steps:

1. Find the factory object;
2. Create the module;
3. Set up its parameters and gate sizes as needed;
4. Tell the (possibly compound) module to recursively create its internal submodules and connections;
5. Schedule activation message(s) for the new simple module(s).

Each step (except for Step 3.) can be done with one line of code.

See the following example, where Step 3 is omitted:

```
// find factory object
cModuleType *moduleType = cModuleType::get("foo.nodes.WirelessNode");

// create (possibly compound) module and build its submodules (if any)
cModule *module = moduleType->create("node", this);
module->finalizeParameters();
module->buildInside();

// create activation message
module->scheduleStart(simTime());
```

If you want to set up parameter values or gate vector sizes (Step 3.), the code goes between the `create()` and `buildInside()` calls:

```
// create
cModuleType *moduleType = cModuleType::get("foo.nodes.WirelessNode");
cModule *module = moduleType->create("node", this);

// set up parameters and gate sizes before we set up its submodules
module->par("address") = ++lastAddress;
module->finalizeParameters();

module->setGateSize("in", 3);
module->setGateSize("out", 3);

// create internals, and schedule it
module->buildInside();
module->scheduleStart(simTime());
```

4.13.4 Deleting Modules

To delete a module dynamically, use `cModule`'s `deleteModule()` member function:

```
module->deleteModule();
```

If the module was a compound module, this involves recursively deleting all its submodules. An `activity()`-based simple module can also delete itself; in that case, the `deleteModule()` call does not return to the caller.

4.13.5 The preDelete() method

When `deleteModule()` is called on a compound module, individual modules under the compound module are notified by calling their `preDelete()` member functions before any change is actually made.

This notification can be quite useful when the compound module contains modules that hold pointers to each other, necessitated by their complex interactions via C++ method calls. With such modules, destruction can be tricky: given a sufficently complex control flow involving cascading cross-module method calls and signal listeners, it is actually quite easy to accidentally invoke a method on a module that has already been deleted at that point, resulting in a crash. (Note that destructors of collaborating modules cannot rely on being invoked in any particular order, because that order is determined factors, e.g. submodule order in NED, which are out the of control of the C++ code.)

`preDelete()` is a `cComponent` virtual method that, similar to `handleMessage()` and `initialize()`, is intended for being overridden by the user. When a compound module is deleted, `deleteModule()` first invokes `preDelete()` recursively on the submodule tree, and only starts deleting modules after that. This gives a chance to modules that override `preDelete()` to set pointers to collaborating modules to `nullptr`, or otherwise ensure that nothing bad will happen once modules start being deleted.

`preDelete()` receives in an argument the pointer of the module on which `deleteModule()` was invoked. This allows the module to tell apart cases when e.g. it is deleted in itself, or as part of a larger unit.

An example:

```
void Foo::preDelete(cComponent *root)
{
    barModule = nullptr;
}
```

4.13.6 Component Weak Pointers

`opp_component_ptr<T>` offers an answer to a related problem: how to detect when a module we have a pointer to is deleted, so that we no longer try to access it.

`opp_component_ptr<T>` is a smart pointer that points to a `cComponent` object (i.e. a module or a channel), and automatically becomes `nullptr` when the referenced object is deleted. It is a non-owning ("weak") pointer, i.e. the pointer going out of scope has no effect on the referenced object.

In practice, one would replace bare pointers in the code (for example, `Foo*`) with `opp_component_ptr<Foo>` smart pointers, and test before accessing the other module that the pointer is still valid.

An example:

```
opp_component_ptr<Foo> fooModule;  // as class member

if (fooModule)
    fooModule->doSomething();

// or: obtain a bare pointer for multiple use
if (Foo *fooPtr = fooModule.get()) {
    fooPtr->doSomething();
```

```
    fooPtr->doSomethingElse();
}
```

4.13.7 Module Deletion and finish()

`finish()` is called for *all* modules at the end of the simulation, no matter how the modules were created. If a module is dynamically deleted before that, `finish()` will not be invoked (`deleteModule()` does not do it). However, you can still manually invoke it before `deleteModule()`.

You can use the `callFinish()` function to invoke `finish()` (It is not a good idea to invoke `finish()` directly). If you are deleting a compound module, `callFinish()` will recursively invoke `finish()` for all submodules, and if you are deleting a simple module from another module, `callFinish()` will do the context switch for the duration of the call. [12]

Example:

```
mod->callFinish();
mod->deleteModule();
```

4.13.8 Creating Connections

Connections can be created using `cGate`'s `connectTo()` method. `connectTo()` should be invoked on the source gate of the connection, and expects the destination gate pointer as an argument. The use of the words *source* and *destination* correspond to the direction of the arrow in NED files.

```
srcGate->connectTo(destGate);
```

`connectTo()` also accepts a channel object (`cChannel*`) as an additional, optional argument. Similarly to modules, channels can be created using their factory objects that have the type `cChannelType`:

```
cGate *outGate, *inGate;
...

// find factory object and create a channel
cChannelType *channelType = cChannelType::get("foo.util.Channel");
cChannel *channel = channelType->create("channel");

// create connecting
outGate->connectTo(inGate, channel);
```

The channel object will be owned by the source gate of the connection, and one cannot reuse the same channel object with several connections.

Instantiating one of the built-in channel types (`cIdealChannel`, `cDelayChannel` or `cDatarateChannel`) is somewhat simpler, because those classes have static `create()` factory functions, and the step of finding the factory object can be spared. Alternatively, one can use `cChannelType`'s `createIdealChannel()`, `createDelayChannel()` and `createDatarateChannel()` static methods.

[12] The `finish()` function has even been made `protected` in `cSimpleModule`, in order to discourage its invocation from other modules.

The channel object may need to be parameterized before using it for a connection. For example, `cDelayChannel` has a `setDelay()` method, and `cDatarateChannel` has `setDelay()`, `setDatarate()`, `setBitErrorRate()` and `setPacketErrorRate()`.

An example that sets up a channel with a datarate and a delay between two modules:

```
cDatarateChannel *datarateChannel = cDatarateChannel::create("channel");
datarateChannel->setDelay(0.001);
datarateChannel->setDatarate(1e9);
outGate->connectTo(inGate, datarateChannel);
```

Finally, here is a more complete example that creates two modules and connects them in both directions:

```
cModuleType *moduleType = cModuleType::get("TicToc");
cModule *a = modtype->createScheduleInit("a", this);
cModule *b = modtype->createScheduleInit("b", this);

a->gate("out")->connectTo(b->gate("in"));
b->gate("out")->connectTo(a->gate("in"));
```

4.13.9 Removing Connections

The `disconnect()` method of `cGate` can be used to remove connections. This method has to be invoked on the *source* side of the connection. It also destroys the channel object associated with the connection, if one has been set.

```
srcGate->disconnect();
```

4.14 Signals

This section describes *simulation signals*, or signals for short. Signals are a versatile concept that first appeared in OMNEST 4.1.

Simulation signals can be used for:

- exposing statistical properties of the model, without specifying whether and how to record them
- receiving notifications about simulation model changes at runtime, and acting upon them
- implementing a publish-subscribe style communication among modules; this is advantageous when the producer and consumer of the information do not know about each other, and possibly there is many-to-one or many-to-many relationship among them
- emitting information for other purposes, for example as input for custom animation effects

Signals are emitted by components (modules and channels). Signals propagate on the module hierarchy up to the root. At any level, one can register listeners, that is, objects with callback methods. These listeners will be notified (their appropriate methods called) whenever a signal

value is emitted. The result of upwards propagation is that listeners registered at a compound module can receive signals from all components in that submodule tree. A listener registered at the system module can receive signals from the whole simulation.

> **NOTE:** A channel's parent is the (compound) module that contains the connection, not the owner of either gate the channel is connected to.

Signals are identified by signal *names* (i.e. strings), but for efficiency, at runtime we use dynamically assigned numeric identifiers (*signal IDs*, typedef'd as `simsignal_t`). The mapping of signal names to signal IDs is global, so all modules and channels asking to resolve a particular signal name will get back the same numeric signal ID.

Listeners can subscribe to signal names or IDs, regardless of their source. For example, if two different and unrelated module types, say `Queue` and `Buffer`, both emit a signal named `"length"`, then a listener that subscribes to `"length"` at some higher compound module will get notifications from both `Queue` and `Buffer` module instances. The listener can still look at the source of the signal if it wants to distinguish the two (it is available as a parameter to the callback function), but the signals framework itself does not have such a feature.

> **NOTE:** Because the component type that emits the signal is not part of the signal's identity, it is advised to choose signal names carefully. A good naming scheme facilitates "merging" of signals that arrive from different sources but mean the same thing, and reduces the chance of collisions between signals that accidentally have the same name but represent different things.

When a signal is emitted, it can carry a *value* with it. There are multiple overloaded versions of the `emit()` method for different data types, and also overloaded `receiveSignal()` methods in listeners. The signal value can be of selected primitive types, or an object pointer; anything that is not feasible to emit as a primitive type may be wrapped into an object, and emitted as such.

Even when the signal value is of a primitive type, it is possible to convey extra information to listeners via an additional *details* object, which an optional argument of `emit()`.

4.14.1 Design Considerations and Rationale

The implementation of signals is based on the following assumptions:

- subscribe/unsubscribe operations are rare compared to `emit()` calls, so it is `emit()` that needs to be efficient
- the signals mechanism is present in every module, so per-module memory overhead must be kept as low as possible
- it is expected that modules and channels will be heavily instrumented with signals, and only a subset of signals will actually be used (will have listeners) in any particular simulation; therefore, the CPU and memory overhead of momentarily unused signals must be as low as possible

These goals have been achieved in the 4.1 version with the following implementation. First, the data structure that used to store listeners in components is dynamically allocated, so if

there are no listeners, the per-component overhead is only the size of the pointer (which will be `nullptr` then).

Second, additionally there are two bitfields in every component that store which one of the first 64 signals (IDs 0..63) have local listeners and listeners in ancestor modules.[13] Using these bitfields, it is possible to determine in constant time for the first 64 signals whether the signal has listeners, so `emit()` can return immediately if there are none. For other signals, `emit()` needs to examine the listener lists up to the root every time. Even if a simulation uses more than 64 signals, in performance-critical situations it is possible to arrange that frequently emitted signals (e.g. `"txBegin"`) get the "fast" signal IDs, while infrequent signals (like e.g. `"routerDown"`) get the rest.

4.14.2 The Signals Mechanism

Signal-related methods are declared on `cComponent`, so they are available for both `cModule` and `cChannel`.

Signal IDs

Signals are identified by names, but internally numeric signal IDs are used for efficiency. The `registerSignal()` method takes a signal name as parameter, and returns the corresponding `simsignal_t` value. The method is static, illustrating the fact that signal names are global. An example:

```
simsignal_t lengthSignalId = registerSignal("length");
```

The `getSignalName()` method (also static) does the reverse: it accepts a `simsignal_t`, and returns the name of the signal as `const char *` (or `nullptr` for invalid signal handles):

```
const char *signalName = getSignalName(lengthSignalId); // --> "length"
```

NOTE: Since OMNEST 4.3, the lifetime of signal IDs is the entire program, and it is possible to call `registerSignal()` from initializers of global variables, e.g. static class members. In earlier versions, signal IDs were usually allocated in `initialize()`, and were only valid for that simulation run.

Emitting Signals

The `emit()` family of functions emit a signal from the module or channel. `emit()` takes a signal ID (`simsignal_t`) and a value as parameters:

```
emit(lengthSignalId, queue.length());
```

The value can be of type `bool`, `long`, `double`, `simtime_t`, `const char *`, or `(const) cObject *`. Other types can be cast into one of these types, or wrapped into an object subclassed from `cObject`.

`emit()` also has an extra, optional object pointer argument named `details`, with the type `cObject*`. This argument may be used to convey to listeners extra information.[14]

[13] It is assumed that there will be typically less than 64 frequently used signals used at a time in a simulation.
[14] The *details* parameter was added in OMNEST 5.0.

When there are no listeners, the runtime cost of `emit()` is usually minimal. However, if producing a value has a significant runtime cost, then the `mayHaveListeners()` or `hasListeners()` method can be used to check beforehand whether the given signal has any listeners at all – if not, producing the value and emitting the signal can be skipped.

Example usage:

```
if (mayHaveListeners(distanceToTargetSignal)) {
    double d = sqrt((x-targetX)*(x-targetX) + (y-targetY)*(y-targetY));
    emit(distanceToTargetSignal, d);
}
```

The `mayHaveListeners()` method is very efficient (a constant-time operation), but may return false positive. In contrast, `hasListeners()` will search up to the top of the module tree if the answer is not cached, so it is generally slower. We recommend that you take into account the cost of producing notification information when deciding between `mayHaveListeners()` and `hasListeners()`.

Signal Declarations

Since OMNEST 4.4, signals can be declared in NED files for documentation purposes, and OMNEST can check that only declared signals are emitted, and that they actually conform to the declarations (with regard to the data type, etc.)

The following example declares a queue module that emits a signal named `queueLength`:

```
simple Queue
{
    parameters:
        @signal[queueLength](type=long);
        ...
}
```

Signals are declared with the `@signal` property on the module or channel that emits it. (NED properties are described in 3.12). The property index corresponds to the signal name, and the property's body may declare various attributes of the signal; currently only the data type is supported.

The `type` property key is optional; when present, its value should be `bool`, `long`, `unsigned long`, `double`, `simtime_t`, `string`, or a registered class name optionally followed by a question mark. Classes can be registered using the `Register_Class()` or `Register_Abstract_Class()` macros; these macros create a `cObjectFactory` instance, and the simulation kernel will call `cObjectFactory`'s `isInstance()` method to check that the emitted object is really a subclass of the declared class. `isInstance()` just wraps a C++ `dynamic_cast`.)

A question mark after the class name means that the signal is allowed to emit `nullptr` pointers. For example, a module named `PPP` may emit the frame (packet) object every time it starts transmiting, and emit `nullptr` when the transmission is completed:

```
simple PPP
{
    parameters:
        @signal[txFrame](type=PPPFrame?);  // a PPPFrame or nullptr
        ...
}
```

The property index may contain wildcards, which is important for declaring signals whose names are only known at runtime. For example, if a module emits signals called `session-1-seqno`, `session-2-seqno`, `session-3-seqno`, etc., those signals can be declared as:

```
    @signal[session-*-seqno]();
```

Enabling/Disabling Signal Checking

Starting with OMNEST 5.0, signal checking is turned on by default when the simulation kernel is compiled in debug mode, requiring all signals to be declared with `@signal`. (It is turned off in release mode simulation kernels due to performance reasons.)

If needed, signal checking can be disabled with the **`check-signals`** configuration option:

```
check-signals = false
```

Signal Data Objects

When emitting a signal with a `cObject*` pointer, you can pass as data an object that you already have in the model, provided you have a suitable object at hand. However, it is often necessary to declare a custom class to hold all the details, and fill in an instance just for the purpose of emitting the signal.

The custom notification class must be derived from `cObject`. We recommend that you also add `noncopyable` as a base class, because then you don't need to write a copy constructor, assignment operator, and `dup()` function, sparing some work. When emitting the signal, you can create a temporary object, and pass its pointer to the `emit()` function.

An example of custom notification classes are the ones associated with model change notifications (see 4.14.3). For example, the data class that accompanies a signal that announces that a gate or gate vector is about to be created looks like this:

```
class cPreGateAddNotification : public cObject, noncopyable
{
  public:
    cModule *module;
    const char *gateName;
    cGate::Type gateType;
    bool isVector;
};
```

And the code that emits the signal:

```
if (hasListeners(PRE_MODEL_CHANGE))
{
    cPreGateAddNotification tmp;
    tmp.module = this;
    tmp.gateName = gatename;
    tmp.gateType = type;
    tmp.isVector = isVector;
    emit(PRE_MODEL_CHANGE, &tmp);
}
```

Subscribing to Signals

The `subscribe()` method registers a listener for a signal. Listeners are objects that extend the `cIListener` class. The same listener object can be subscribed to multiple signals. `subscribe()` has two arguments: the signal and a pointer to the listener object:

```
cIListener *listener = ...;
simsignal_t lengthSignalId = registerSignal("length");
subscribe(lengthSignalId, listener);
```

For convenience, the `subscribe()` method has a variant that takes the signal name directly, so the `registerSignal()` call can be omitted:

```
cIListener *listener = ...;
subscribe("length", listener);
```

One can also subscribe at other modules, not only the local one. For example, in order to get signals from all parts of the model, one can subscribe at the system module level:

```
cIListener *listener = ...;
getSimulation()->getSystemModule()->subscribe("length", listener);
```

The `unsubscribe()` method has the same parameter list as `subscribe()`, and unregisters the given listener from the signal:

```
unsubscribe(lengthSignalId, listener);
```

or

```
unsubscribe("length", listener);
```

It is an error to subscribe the same listener to the same signal twice.

It is possible to test whether a listener is subscribed to a signal, using the `isSubscribed()` method which also takes the same parameter list.

```
if (isSubscribed(lengthSignalId, listener)) {
    ...
}
```

For completeness, there are methods for getting the list of signals that the component has subscribed to (`getLocalListenedSignals()`), and the list of listeners for a given signal (`getLocalSignalListeners()`). The former returns `std::vector<simsignal_t>`; the latter takes a signal ID (`simsignal_t`) and returns `std::vector<cIListener*>`.

The following example prints the number of listeners for each signal:

```
EV << "Signal listeners:\n";
std::vector<simsignal_t> signals = getLocalListenedSignals();
for (unsigned int i = 0; i < signals.size(); i++) {
    simsignal_t signalID = signals[i];
    std::vector<cIListener*> listeners = getLocalSignalListeners(signalID);
    EV << getSignalName(signalID) << ": " << listeners.size() << " signals\n";
}
```

Listeners

Listeners are objects that subclass from the `cIListener` class, which declares the following methods:

```
class cIListener
{
  public:
    virtual ~cIListener() {}
    virtual void receiveSignal(cComponent *src, simsignal_t id,
                               bool value, cObject *details) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id,
                               intval_t value, cObject *details) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id,
                               uintval_t value, cObject *details) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id,
                               double value, cObject *details) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id,
                               simtime_t value, cObject *details) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id,
                               const char *value, cObject *details) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id,
                               cObject *value, cObject *details) = 0;
    virtual void finish(cComponent *component, simsignal_t id) {}
    virtual void subscribedTo(cComponent *component, simsignal_t id) {}
    virtual void unsubscribedFrom(cComponent *component, simsignal_t id) {}
};
```

This class has a number of virtual methods:

- Several overloaded `receiveSignal()` methods, one for each data type. Whenever a signal is emitted (via `emit()`), the matching `receiveSignal()` method is invoked on the subscribed listeners.
- `finish()` is called by a component on its local listeners after the component's `finish()` method was called. If the listener is subscribed to multiple signals or at multiple components, the method will be called multiple times. Note that `finish()` methods in general are not invoked if the simulation terminates with an error, so that method is not a place for doing cleanup.
- `subscribedTo()`, `unsubscribedFrom()` are called when this listener object is subscribed/unsubscribed to (from) a signal. These methods give the opportunity for listeners to track whether and where they are subscribed. It is also OK for a listener to delete itself in the last statement of the `unsubscribedFrom()` method, but you must be sure that there are no other places the same listener is still subscribed.

Since `cIListener` has a large number of pure virtual methods, it is more convenient to subclass from `cListener`, a do-nothing implementation instead. It defines `finish()`, `subscribedTo()` and `unsubscribedFrom()` with an empty body, and the `receiveSignal()` methods with a bodies that throw a `"Data type not supported"` error. You can redefine the `receiveSignal()` method(s) whose data type you want to support, and signals emitted with other (unexpected) data types will result in an error instead of going unnoticed.

The order in which listeners will be notified is undefined (it is not necessarily the same order in which listeners were subscribed.)

Listener Life Cycle

When a component (module or channel) is deleted, it automatically unsubscribes (but does not delete) the listeners it has. When a module is deleted, it first unsubscribes all listeners from all modules and channels in its submodule tree before starting to recursively delete the modules and channels themselves.

When a listener is deleted, it automatically unsubscribes from all components it is subscribed to.[15]

> **NOTE:** If your module has added listeners to other modules (e.g. the toplevel module), these listeners must be unsubscribed in the module destructor at latest. Remember to make sure the modules still exist before you call `unsubscribe()` on them, unless they are an ancestor of your module in the module tree.

4.14.3 Listening to Model Changes

In simulation models it is often useful to hold references to other modules, a connecting channel or other objects, or to cache information derived from the model topology. However, such pointers or data may become invalid when the model changes at runtime, and need to be updated or recalculated. The problem is how to get notification that something has changed in the model.

> **NOTE:** Whenever you see a `cModule*`, `cChannel*`, `cGate*` or similar pointer kept as state in a simple module, you should think about how it will be kept up-to-date if the model changes at runtime.

The solution is, of course, signals. OMNEST has two built-in signals, `PRE_MODEL_CHANGE` and `POST_MODEL_CHANGE` (these macros are `simsignal_t` values, not names) that are emitted before and after each model change.

Pre/post model change notifications are emitted with data objects that carry the details of the change. The data classes are:

- `cPreModuleAddNotification` / `cPostModuleAddNotification`
- `cPreModuleDeleteNotification` / `cPostModuleDeleteNotification`
- `cPreModuleReparentNotification` / `cPostModuleReparentNotification`
- `cPreGateAddNotification` / `cPostGateAddNotification`
- `cPreGateDeleteNotification` / `cPostGateDeleteNotification`
- `cPreGateVectorResizeNotification` / `cPostGateVectorResizeNotification`
- `cPreGateConnectNotification` / `cPostGateConnectNotification`
- `cPreGateDisconnectNotification` / `cPostGateDisconnectNotification`
- `cPrePathCreateNotification` / `cPostPathCreateNotification`
- `cPrePathCutNotification` / `cPostPathCutNotification`

[15] This behavior is new in OMNEST6.0. Prior versions mandated that the listener be already unsubscribed from all places when its destructor runs, but did not automatically unsubscribe.

- `cPreParameterChangeNotification` / `cPostParameterChangeNotification`
- `cPreDisplayStringChangeNotification` / `cPostDisplayStringChangeNotification`

They all subclass from `cModelChangeNotification`, which is of course a `cObject`. Inside the listener, you can use `dynamic_cast<>` to figure out what notification arrived.

> **NOTE:** Please look up these classes in the API documentation to see their data fields, when exactly they get fired, and what one needs to be careful about when using them.

An example listener that prints a message when a module is deleted:

```
class MyListener : public cListener
{
   ...
};

void MyListener::receiveSignal(cComponent *src, simsignal_t id, cObject *value,
                               cObject *details)
{
    if (dynamic_cast<cPreModuleDeleteNotification *>(value)) {
        cPreModuleDeleteNotification *data = (cPreModuleDeleteNotification *)value
        EV << "Module " << data->module->getFullPath() << " is about to be deleted
    }
}
```

If you'd like to get notification about the deletion of any module, you need to install the listener on the system module:

```
getSimulation()->getSystemModule()->subscribe(PRE_MODEL_CHANGE, listener);
```

> **NOTE:** `PRE_MODEL_CHANGE` and `POST_MODEL_CHANGE` are fired on the module (or channel) affected by the change, and *not* on the module which executes the code that causes the change. For example, *pre-module-deleted* is fired on the module to be removed, and *post-module-deleted* is fired on its parent (because the original module no longer exists), and not on the module that contains the `deleteModule()` call.

> **NOTE:** A listener will *not* receive *pre/post-module-deleted* notifications if the whole submodule tree that contains the subscription point is deleted. This is because compound module destructors begin by unsubscribing all modules/channels in the subtree before starting recursive deletion.

4.15 Signal-Based Statistics Recording

4.15.1 Motivation

One use of signals is to expose variables for result collection without telling where, how, and whether to record them. With this approach, modules only publish the variables, and the actual result recording takes place in listeners. Listeners may be added by the simulation

framework (based on the configuration), or by other modules (for example by dedicated result collection modules).

The signals approach allows for several possibilities:

- Provides a controllable level of detail: in some simulation runs you may want to record all values as a time series, in other runs only record the mean, time average, minimum/-maximum value, standard deviation etc, and in yet other runs you may want to record the distribution as a histogram;
- Depending on the purpose of the simulation experiment, you may want to process the results before recording them, for example record a smoothed or filtered value, record the percentage of time the value is nonzero or over a threshold, record the sum of the values, etc.;
- You may want aggregate statistics, e.g. record the total number of packet drops or the average end-to-end delay for the whole network;
- You may want to record combined statistics, for example a drop percentage (drop count/total number of packets);
- You may want to ignore results generated during the warm-up period or during other transients.

With the signals approach the above goals can be fulfilled.

4.15.2 Declaring Statistics

Introduction

In order to record simulation results based on signals, one must add `@statistic` properties to the simple module's (or channel's) NED definition. A `@statistic` property defines the name of the statistic, which signal(s) are used as input, what processing steps are to be applied to them (e.g. smoothing, filtering, summing, differential quotient), and what properties are to be recorded (minimum, maximum, average, etc.) and in which form (vector, scalar, histogram). Record items can be marked optional, which lets you denote a *"default"* and a more comprehensive *"all"* result set to be recorded; the list of record items can be further tweaked from the configuration. One can also specify a descriptive name ("title") for the statistic, and also a measurement unit.

The following example declares a queue module with a queue length statistic:

```
simple Queue
{
    parameters:
        @statistic[queueLength](record=max,timeavg,vector?);
    gates:
        input in;
        output out;
}
```

As you can see, statistics are represented with indexed NED properties (see 3.12). The property name is always `statistic`, and the index (here, `queueLength`) is the name of the statistic. The property value, that is, everything inside the parentheses, carries hints and extra information for recording.

The above `@statistic` declaration assumes that module's C++ code emits the queue's updated length as signal `queueLength` whenever elements are inserted into the queue or are removed from it. By default, the maximum and the time average of the queue length will be recorded as scalars. One can also instruct the simulation (or parts of it) to record "all" results; this will turn on optional record items, those marked with a question mark, and then the queue lengths will also be recorded into an output vector.

> **NOTE:** The configuration lets you fine-tune the list of result items even beyond the `default` and `all` settings; see section 12.2.3.

In the above example, the signal to be recorded was taken from the statistic name. When that is not suitable, the `source` property key lets you specify a different signal as input for the statistic. The following example assumes that the C++ code emits a `qlen` signal, and declares a `queueLength` statistic based on that:

```
simple Queue
{
    parameters:
        @signal[qlen](type=int); // optional
        @statistic[queueLength](source=qlen; record=max,timeavg,vector?);
        ...
}
```

Note that beyond the `source=qlen` property key we have also added a signal declaration (`@signal` property) for the `qlen` signal. Declaring signals is currently optional and in fact `@signal` properties are currently ignored by the system, but it is a good practice nevertheless.

It is also possible to apply processing to a signal before recording it. Consider the following example:

```
@statistic[dropCount](source=count(drop); record=last,vector?);
```

This records the total number of packet drops as a scalar, and optionally the number of packets dropped in the function of time as a vector, provided the C++ code emits a `drop` signal every time a packet is dropped. The value and even the data type of the `drop` signal is indifferent, because only the number of emits will be counted. Here, `count()` is a *result filter*.

> **NOTE:** Starting from OMNEST 4.4, items containing parens (e.g. `count(drop)`) no longer need to be enclosed in quotation marks.

Another example:

```
@statistic[droppedBytes](source=sum(packetBytes(pkdrop)); record=last,
vector?);
```

This example assumes that the C++ code emits a `pkdrop` signal with a packet (`cPacket*` pointer) as a value. Based on that signal, it records the total number of bytes dropped (as a scalar, and optionally as a vector too). The `packetBytes()` filter extracts the number of bytes from each packet using `cPacket`'s `getByteLength()` method, and the `sum()` filter, well, sums them up.

Arithmetic expressions can also be used. For example, the following line computes the number of dropped bytes using the `packetBits()` filter.

```
@statistic[droppedBytes](source=sum(8*packetBits(pkdrop)); record=last,
vector?);
```

The source can also combine multiple signals in an arithmetic expression:

```
@statistic[dropRate](source=count(drop)/count(pk); record=last,vector?);
```

When multiple signals are used, a value arriving on either signal will result in one output value. The computation will use the last values of the other signals (sample-hold interpolation). One limitation regarding multiple signals is that the same signal cannot occur twice, because it would cause glitches in the output.

Record items may also be expressions and contain filters. For example, the statistic below is functionally equivalent to one of the above examples: it also computes and records as scalar and as vector the total number of bytes dropped, using a `cPacket*`-valued signal as input; however, some of the computations have been shifted into the recorder part.

```
@statistic[droppedBytes](source=packetBits(pkdrop); record=last(8*sum),
vector(8*sum)?);
```

Property Keys

The following keys are understood in `@statistic` properties:

source : Defines the input for the recorders (see `record=` key). When missing, the statistic name is taken as the signal name;

record : Contains a list of recording modes, separated by comma. Recording modes define how to record the source (see `source=` key).

title : A longer, descriptive name for the statistic signal; result visualization tools may use it as chart label, e.g. in the legend.

unit : Measurement unit of the values. This may also appear in charts.

interpolationmode : Defines how to interpolate signal values where needed (e.g. for drawing); possible values are `none`, `sample-hold`, `backward-sample-hold`, `linear`.

enum : Defines symbolic names for various integer signal values. The property value must be a string, containing *name=value* pairs separated by comma. Example: `"IDLE=1,BUSY=2,DOWN=3"`.

Available Filters and Recorders

The following table contains the list of predefined result filters. All filters in the table output a value for each input value.

Filter	Description
`count`	Computes and outputs the count of values received so far.
`sum`	Computes and outputs the sum of values received so far.
`min`	Computes and outputs the minimum of values received so far.
`max`	Computes and outputs the maximum of values received so far.

`mean`	Computes and outputs the average (sum / count) of values received so far.
`timeavg`	Regards the input values and their timestamps as a step function (sample-hold style), and computes and outputs its time average (integral divided by duration).
`constant0`	Outputs a constant 0 for each received value (independent of the value).
`constant1`	Outputs a constant 1 for each received value (independent of the value).
`packetBits`	Expects `cPacket` pointers as value, and outputs the bit length for each received one. Non-`cPacket` values are ignored.
`packetBytes`	Expects `cPacket` pointers as value, and outputs the byte length for each received one. Non-`cPacket` values are ignored.
`sumPerDuration`	For each value, computes the sum of values received so far, divides it by the duration, and outputs the result.
`removeRepeats`	Removes repeated values, i.e. discards values that are the same as the previous value.

The list of predefined result recorders:

Recorder	**Description**
`last`	Records the last value into an output scalar.
`count`	Records the count of the input values into an output scalar; functionally equivalent to `last(count)`
`sum`	Records the sum of the input values into an output scalar (or zero if there was none); functionally equivalent to `last(sum)`
`min`	Records the minimum of the input values into an output scalar (or positive infinity if there was none); functionally equivalent to `last(min)`
`max`	Records the maximum of the input values into an output scalar (or negative infinity if there was none); functionally equivalent to `last(max)`
`mean`	Records the mean of the input values into an output scalar (or NaN if there was none); functionally equivalent to `last(mean)`
`timeavg`	Regards the input values with their timestamps as a step function (sample-hold style), and records the time average of the input values into an output scalar; functionally equivalent to `last(timeavg)`
`stats`	Computes basic statistics (count, mean, std.dev, min, max) from the input values, and records them into the output scalar file as a statistic object.
`histogram`	Computes a histogram and basic statistics (count, mean, std.dev, min, max) from the input values, and records the result into the output scalar file as a histogram object.
`vector`	Records the input values with their timestamps into an output vector.

NOTE: You can have the list of available result filters and result recorders printed by executing the `opp_run -h resultfilters` and `opp_run -h resultrecorders` commands.

Naming and Attributes of Recorded Results

The names of recorded result items will be formed by concatenating the statistic name and the recording mode with a colon between them: "*<statisticName>*:*<recordingMode>*".

Thus, the following statistics

```
@statistic[dropRate](source=count(drop)/count(pk); record=last,vector?);
@statistic[droppedBytes](source=packetBytes(pkdrop); record=sum,vector(sum)?);
```

will produce the following scalars: `dropRate:last`, `droppedBytes:sum`, and the following vectors: `dropRate:vector`, `droppedBytes:vector(sum)`.

All property keys (except for `record`) are recorded as result attributes into the vector file or scalar file. The `title` property will be tweaked a little before recording: the recording mode will be added after a comma, otherwise all result items saved from the same statistic would have exactly the same name.

Example: "Dropped Bytes, sum", "Dropped Bytes, vector(sum)"

It is allowed to use other property keys as well, but they won't be interpreted by the OMNEST runtime or the result analysis tool.

Source and Record Expressions in Detail

To fully understand `source` and `record`, it will be useful to see how result recording is set up.

When a module or channel is created in the simulation, the OMNEST runtime examines the `@statistic` properties on its NED declaration, and adds listeners on the signals they mention as input. There are two kinds of listeners associated with result recording: *result filters* and *result recorders*. Result filters can be chained, and at the end of the chain there is always a recorder. So, there may be a recorder directly subscribed to a signal, or there may be a chain of one or more filters plus a recorder. Imagine it as a pipeline, or rather a "pipe tree", where the tree roots are signals, the leaves are result recorders, and the intermediate nodes are result filters.

Result filters typically perform some processing on the values they receive on their inputs (the previous filter in the chain or directly a signal), and propagate them to their output (chained filters and recorders). A filter may also swallow (i.e. not propagate) values. Recorders may write the received values into an output vector, or record output scalar(s) at the end of the simulation.

Many operations exist both in filter and recorder form. For example, the `sum` *filter* propagates the sum of values received on its input to its output; and the `sum` *recorder* only computes the the sum of received values in order to record it as an output scalar on simulation completion.

The next figure illustrates which filters and recorders are created and how they are connected for the following statistics:

```
@statistic[droppedBits](source=8*packetBytes(pkdrop); record=sum,vector(sum));
```

Figure 4.4: Result filters and recorders chained

HINT: To see how result filters and recorders have been set up for a particular simulation, run the simulation with the **debug-statistics-recording** configuration option, e.g. specify `-debug-statistics-recording=true` on the command line.

4.15.3 Statistics Recording for Dynamically Registered Signals

It is often convenient to have a module record statistics per session, per connection, per client, etc. One way of handling this use case is registering signals dynamically (e.g. `session1-jitter`, `session2-jitter`, ...), and setting up `@statistic`-style result recording on each.

The NED file would look like this:

```
@signal[session*-jitter](type=simtime_t); // note the wildcard
@statisticTemplate[sessionJitter](record=mean,vector?);
```

In the C++ code of the module, you need to register each new signal with `registerSignal()`, and in addition, tell OMNEST to set up statistics recording for it as described by the `@statisticTemplate` property. The latter can be achieved by calling `getEnvir()->addResultRecorders()`.

```
char signalName[32];
sprintf(signalName, "session%d-jitter", sessionNum);
simsignal_t signal = registerSignal(signalName);

char statisticName[32];
sprintf(statisticName, "session%d-jitter", sessionNum);
cProperty *statisticTemplate =
    getProperties()->get("statisticTemplate", "sessionJitter");
getEnvir()->addResultRecorders(this, signal, statisticName, statisticTemplate);
```

In the `@statisticTemplate` property, the `source` key will be ignored (because the signal given as parameter will be used as source). The actual name and index of property will also be ignored. (With `@statistic`, the index holds the result name, but here the name is explicitly specified in the statisticName parameter.)

When multiple signals are recorded using a common `@statisticTemplate` property, you'll want the titles of the recorded statistics to differ for each signal. This can be achieved by using dollar variables in the `title` key of `@statisticTemplate`. The following variables are available:

- `$name`: name of the statistic

- `$component`: component fullpath
- `$mode`: recording mode
- `$namePart[0-9]+`: given part of statistic name, when split along colons (:); numbering starts with 1

For example, if the statistic name is `"conn:host1-to-host4(3):bytesSent"`, and the title is `"bytes sent in connection $namePart2"`, it will become `"bytes sent in connection host1-to-host4(3)"`.

4.15.4 Adding Result Filters and Recorders Programmatically

As an alternative to `@statisticTemplate` and `addResultRecorders()`, it is also possible to set up result recording programmatically, by creating and attaching result filters and recorders to the desired signals.

> **NOTE:** It is important to know that `@statistic` implements warmup period support by including a special *warmup period filter* at the front of the filter/recorder chain. When adding result filters and recorders manually, you need to add this filter manually as well.

The following code example sets up recording to an output vector after removing duplicate values, and is essentially equivalent to the following `@statistic` line:

```
@statistic[queueLength](source=qlen; record=vector(removeRepeats);
                        title="Queue Length"; unit=packets);
```

The C++ code:

```
simsignal_t signal = registerSignal("qlen");

cResultFilter *warmupFilter =
        cResultFilterType::get("warmup")->create();
cResultFilter *removeRepeatsFilter =
        cResultFilterType::get("removeRepeats")->create();
cResultRecorder *vectorRecorder =
        cResultRecorderType::get("vector")->create();
opp_string_map *attrs = new opp_string_map;
(*attrs)["title"] = "Queue Length";
(*attrs)["unit"] = "packets";
cResultRecorder::Context ctx { this, "queueLength", "vector",
                               nullptr, attrs};
vectorRecorder->init(&ctx);

subscribe(signal, warmupFilter);
warmupFilter->addDelegate(removeRepeatsFilter);
removeRepeatsFilter->addDelegate(vectorRecorder);
```

4.15.5 Emitting Signals

Emitting signals for statistical purposes does not differ much from emitting signals for any other purpose. Statistic signals are primarily expected to contain numeric values, so the

overloaded `emit()` functions that take `long`, `double` and `simtime_t` are going to be the most useful ones.

Emitting with timestamp. The emitted values are associated with the current simulation time. At times it might be desirable to associate them with a different timestamp, in much the same way as the `recordWithTimestamp()` method of `cOutVector` (see 7.10.1) does. For example, assume that you want to emit a signal at the start of every successful wireless frame reception. However, whether any given frame reception is going to be successful can only be known after the reception has completed. Hence, values can only be emitted at reception completion, and need to be associated with past timestamps.

To emit a value with a different timestamp, an object containing a $(timestamp, value)$ pair needs to be filled in, and emitted using the `emit(simsignal_t, cObject *)` method. The class is called `cTimestampedValue`, and it simply has two public data members called `time` and `value`, with types `simtime_t` and `double`. It also has a convenience constructor taking these two values.

> **NOTE:** `cTimestampedValue` is not part of the signal mechanism. Instead, the result recording listeners provided by OMNEST have been written in a way so that they understand `cTimestampedValue`, and know how to handle it.

An example usage:

```
simtime_t frameReceptionStartTime = ...;
double receivePower = ...;
cTimestampedValue tmp(frameReceptionStartTime, receivePower);
emit(recvPowerSignal, &tmp);
```

If performance is critical, the `cTimestampedValue` object may be made a class member or a static variable to eliminate object construction/destruction time.[16]

Timestamps must be monotonically increasing.

Emitting non-numeric values. Sometimes it is practical to have multi-purpose signals, or to retrofit an existing non-statistical signal so that it can be recorded as a result. For this reason, signals having non-numeric types (that is, `const char *` and `cObject *`) may also be recorded as results. Wherever such values need to be interpreted as numbers, the following rules are used by the built-in result recording listeners:

- Strings are recorded as 1.0;
- Objects that can be cast to `cITimestampedValue` are recorded using the `getSignalTime()` and `getSignalValue()` methods of the class;
- Other objects are recorded as 1.0, except for `nullptr` which is recorded as 0.0.

`cITimestampedValue` is a C++ interface that may be used as an additional base class for any class. It is declared like this:

```
class cITimestampedValue {
    public:
        virtual ~cITimestampedValue() {}
        virtual double getSignalValue(simsignal_t signalID) = 0;
        virtual simtime_t getSignalTime(simsignal_t signalID);
};
```

[16] It is safe to use a static variable here because the simulation program is single-threaded, but ensure that there isn't a listener somewhere that would modify the same static variable during firing.

`getSignalValue()` is pure virtual (it must return some value), but `getSignalTime()` has a default implementation that returns the current simulation time. Note the `signalID` argument that allows the same class to serve multiple signals (i.e. to return different values for each).

4.15.6 Writing Result Filters and Recorders

You can define your own result filters and recorders in addition to the built-in ones. Similar to defining modules and new NED functions, you have to write the implementation in C++, and then register it with a registration macro to let OMNEST know about it. The new result filter or recorder can then be used in the `source=` and `record=` attributes of `@statistic` properties just like the built-in ones.

Result filters must be subclassed from `cResultFilter` or from one of its more specific subclasses `cNumericResultFilter` and `cObjectResultFilter`. The new result filter class needs to be registered using the `Register_ResultFilter(NAME, CLASSNAME)` macro.

Similarly, a result recorder must subclass from the `cResultRecorder` or the more specific `cNumericResultRecorder` class, and be registered using the `Register_ResultRecorder(NAME, CLASSNAME)` macro.

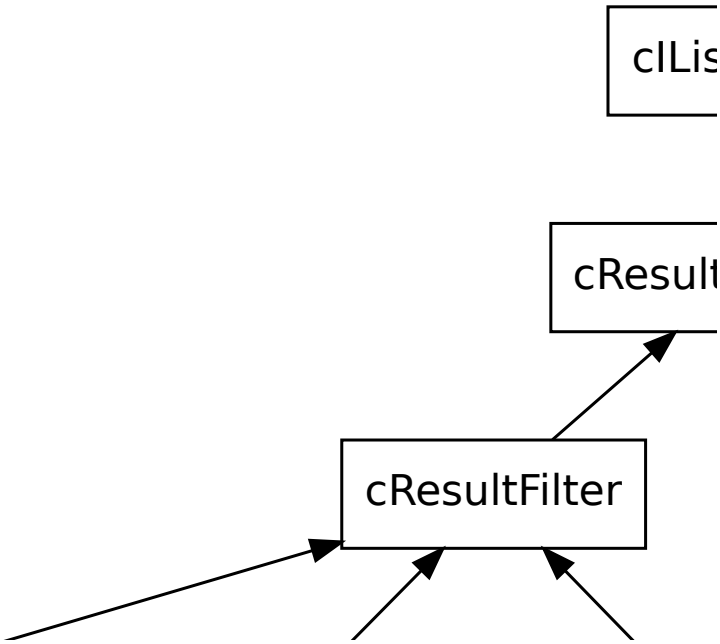


Figure 4.5: Inheritance of result filter and recorder classes

An example result filter implementation from the simulation runtime:

```
/**
 * Filter that outputs the sum of signal values divided by the measurement
 * interval (simtime minus warmup period).
 */
class SumPerDurationFilter : public cNumericResultFilter
{
```

```
    protected:
        double sum;
    protected:
        virtual bool process(simtime_t& t, double& value, cObject *details);
    public:
        SumPerDurationFilter() {sum = 0;}
};

Register_ResultFilter("sumPerDuration", SumPerDurationFilter);

bool SumPerDurationFilter::process(simtime_t& t, double& value, cObject *)
{
    sum += value;
    value = sum / (simTime() - getSimulation()->getWarmupPeriod());
    return true;
}
```

Chapter 5

Messages and Packets

5.1 Overview

Messages are a central concept in OMNEST. In the model, message objects represent events, packets, commands, jobs, customers or other kinds of entities, depending on the model domain.

Messages are represented with the `cMessage` class and its subclass `cPacket`. `cPacket` is used for network packets (frames, datagrams, transport packets, etc.) in a communication network, and `cMessage` is used for everything else. Users are free to subclass both `cMessage` and `cPacket` to create new types and to add data.

`cMessage` has the following fields; some are used by the simulation kernel, and others are provided for the convenience of the simulation programmer:

- The *name* field is a string (`const char *`), which can be freely used by the simulation programmer. The message name is displayed at many places in the graphical runtime interface, so it is generally useful to choose a descriptive name. Message name is inherited from `cObject` (see section 7.1.2).
- *Message kind* is an integer field. Some negative values are reserved by the simulation library, but zero and positive values can be freely used in the model for any purpose. Message kind is typically used to carry a value that conveys the role, type, category or identity of the message.
- The *scheduling priority* field is used by the simulation kernel to determine the delivery order of messages that have the same arrival time values. This field is rarely used in practice.
- The *send time*, *arrival time*, *source module*, *source gate*, *destination module*, *destination gate* fields store information about the message's last sending or scheduling, and should not be modified from the model. These fields are primarily used internally by the simulation kernel while the message is in the future events set (FES), but the information is still in the message object when the message is delivered to a module.
- *Time stamp* (not to be confused with *arrival time*) is a utility field, which the programmer can freely use for any purpose. The time stamp is not examined or changed by the simulation kernel at all.

- The *parameter list*, *control info* and *context pointer* fields make some simulation tasks easier to program, and they will be discussed later.

The `cPacket` class extends `cMessage` with fields that are useful for representing network packets:

- The *packet length* field represents the length of the packet in bits. It is used by the simulation kernel to compute the transmission duration when a packet travels through a connection that has an assigned data rate, and also for error modeling on channels with a nonzero bit error rate.
- The *encapsulated packet* field helps modeling protocol layers by supporting the concept of encapsulation and decapsulation.
- The *bit error flag* field carries the result of error modelling after the packet is sent through a channel that has a nonzero packet error rate (PER) or bit error rate (BER). It is up to the receiver to examine this flag after having received the packet, and to act upon it.
- The *duration* field carries the transmission duration after the packet was sent through a channel with a data rate.
- The *is-reception-start* flag tells whether this packet represents the start or the end of the reception after the packet travelled through a channel with a data rate. This flag is controlled by the *deliver-on-reception-start* flag of the receiving gate.

5.2 The cMessage Class

5.2.1 Basic Usage

The `cMessage` constructor accepts an *object name* and a *message kind*, both optional:

```
cMessage(const char *name=nullptr, short kind=0);
```

Descriptive message names can be very useful when tracing, debugging or demonstrating the simulation, so it is recommended to use them. Message kind is usually initialized with a symbolic constant (e.g. an *enum* value) which signals what the message object represents. Only positive values and zero can be used – negative values are reserved for use by the simulation kernel.

The following lines show some examples of message creation:

```
cMessage *msg1 = new cMessage();
cMessage *msg2 = new cMessage("timeout");
cMessage *msg3 = new cMessage("timeout", KIND_TIMEOUT);
```

Once a message has been created, its basic data members can be set with the following methods:

```
void setName(const char *name);
void setKind(short k);
void setTimestamp();
void setTimestamp(simtime_t t);
void setSchedulingPriority(short p);
```

The argument-less `setTimeStamp()` method is equivalent to `setTimeStamp(simTime())`.

The corresponding getter methods are:

```
const char *getName() const;
short getKind() const;
simtime_t getTimestamp() const;
short getSchedulingPriority() const;
```

The `getName()`/`setName()` methods are inherited from a generic base class in the simulation library, `cNamedObject`.

Two more interesting methods:

```
bool isPacket() const;
simtime_t getCreationTime() const;
```

The `isPacket()` method returns `true` if the particular message object is a subclass of `cPacket`, and `false` otherwise. As `isPacket()` is implemented as a virtual function that just contains a `return false` or a `return true` statement, it might be faster than calling `dynamic_cast<cPacket*>`.

The `getCreationTime()` method returns the creation time of the message. It is worthwhile to mention that with cloned messages (see `dup()` later), the creation time of the original message is returned and not the time of the cloning operation. This is particularly useful when modeling communication protocols, because many protocols clone the transmitted packages to be able to do retransmissions and/or segmentation/reassembly.

5.2.2 Duplicating Messages

It is often necessary to duplicate a message or a packet, for example, to send one and keep a copy. Duplication can be done in the same way as for any other OMNEST object:

```
cMessage *copy = msg->dup();
```

The resulting message (or packet) will be an exact copy of the original including message parameters and encapsulated messages, except for the message ID field. The creation time field is also copied, so for cloned messages `getCreationTime()` will return the creation time of the original, not the time of the cloning operation. [1]

When subclassing `cMessage` or `cPacket`, one needs to reimplement `dup()`. The recommended implementation is to delegate to the copy constructor of the new class:

```
class FooMessage : public cMessage {
  public:
    FooMessage(const FooMessage& other) {...}
    virtual FooMessage *dup() const {return new FooMessage(*this);}
    ...
};
```

For generated classes (chapter 6), this is taken care of automatically.

[1] Note, however, that the simulation library may delay the duplication of the encapsulated message until it is really needed; see section 5.4.5.

5.2.3 Message IDs

Every message object has a unique numeric *message ID*. It is normally used for identifying the message in a recorded event log file, but may occasionally be useful for other purposes as well. When a message is cloned (`msg->dup()`), the clone will have a different ID.

There is also another ID called *tree ID*. The tree ID is initialized to the message ID. However, when a message is cloned, the clone will retain the tree ID of the original. Thus, messages that have been created by cloning the same message or its clones will have the same tree ID. Message IDs are of the type `long`, which is is usually enough so that IDs remain unique during the simulation run (i.e. the counter does not wrap).

The methods for obtaining message IDs:

```
long getId() const;
long getTreeId() const;
```

5.2.4 Control Info

One of the main application areas of OMNEST is the simulation of telecommunication networks. Here, protocol layers are usually implemented as modules which exchange packets. Packets themselves are represented by messages subclassed from `cPacket`.

However, communication between protocol layers requires sending additional information to be attached to packets. For example, a TCP implementation sending down a TCP packet to IP will want to specify the destination IP address and possibly other parameters. When IP passes up a packet to TCP after decapsulation from the IP header, it will want to let TCP know at least the source IP address.

This additional information is represented by *control info* objects in OMNEST. Control info objects have to be subclassed from `cObject` (a small footprint base class with no data members), and can be attached to any message. `cMessage` has the following methods for this purpose:

```
void setControlInfo(cObject *controlInfo);
cObject *getControlInfo() const;
cObject *removeControlInfo();
```

When a "command" is associated with the message sending (such as TCP OPEN, SEND, CLOSE, etc), the message kind field (`getKind()`, `setKind()` methods of `cMessage`) should carry the command code. When the command doesn't involve a data packet (e.g. TCP CLOSE command), a dummy packet (empty `cMessage`) can be sent.

An object set as control info via `setControlInfo()` will be owned by the message object. When the message is deallocated, the control info object is deleted as well.

5.2.5 Information About the Last Arrival

The following methods return the sending and arrival times that correspond to the last sending of the message.

```
simtime_t getSendingTime() const;
simtime_t getArrivalTime() const;
```

The following methods can be used to determine where the message came from and which gate it arrived on (or will arrive if it is currently scheduled or under way.) There are two sets of methods, one returning module/gate Ids, and the other returning pointers.

```
int getSenderModuleId() const;
int getSenderGateId() const;
int getArrivalModuleId() const;
int getArrivalGateId() const;
cModule *getSenderModule() const;
cGate *getSenderGate() const;
cModule *getArrivalModule() const;
cGate *getArrivalGate() const;
```

There are further convenience functions to tell whether the message arrived on a specific gate given with id or with name and index.

```
bool arrivedOn(int gateId) const;
bool arrivedOn(const char *gatename) const;
bool arrivedOn(const char *gatename, int gateindex) const;
```

5.2.6 Display String

Display strings affect the message's visualization in graphical user interfaces like Qtenv. Message objects do not store a display string by default, but contain a `getDisplayString()` method that can be overridden in subclasses to return the desired string. The method:

```
const char *getDisplayString() const;
```

Since OMNEST version 5.1, `cPacket`'s default `getDisplayString()` implementation is such so that a packet "inherits" the display string of its encapsulated packet, provided it has one. Thus, in the model of a network stack, the appearance of e.g. an application layer packet will be preserved even after multiple levels of encapsulation.

See section for more information on message display string syntax and possibilities.

5.3 Self-Messages

5.3.1 Using a Message as Self-Message

Messages are often used to represent events internal to a module, such as a periodically firing timer to represent expiry of a timeout. A message is termed *self-message* when it is used in such a scenario – otherwise self-messages are normal messages of class `cMessage` or a class derived from it.

When a message is delivered to a module by the simulation kernel, the `isSelfMessage()` method can be used to determine if it is a self-message; that is, whether it was scheduled with `scheduleAt()`, or sent with one of the `send...()` methods. The `isScheduled()` method returns true if the message is currently scheduled. A scheduled message can also be cancelled (`cancelEvent()`).

```
bool isSelfMessage() const;
bool isScheduled() const;
```

The methods `getSendingTime()` and `getArrivalTime()` are also useful with self-messages: they return the time the message was scheduled and arrived (or will arrive; while the message is scheduled, arrival time is the time it will be delivered to the module).

5.3.2 Context Pointer

`cMessage` contains a *context pointer* of type `void*`, which can be accessed by the following functions:

```
void setContextPointer(void *p);
void *getContextPointer() const;
```

The context pointer can be used for any purpose by the simulation programmer. It is not used by the simulation kernel, and it is treated as a mere pointer (no memory management is done on it).

Intended purpose: a module which schedules several self-messages (timers) will need to identify a self-message when it arrives back to the module, ie. the module will have to determine which timer went off and what to do then. The context pointer can be made to point at a data structure kept by the module which can carry enough "context" information about the event.

5.4 The cPacket Class

5.4.1 Basic Usage

The `cPacket` constructor is similar to the `cMessage` constructor, but it accepts an additional *bit length* argument:

```
cPacket(const char *name=nullptr, short kind=0, int64 bitLength=0);
```

The most important field `cPacket` has over `cMessage` is the message length. This field is kept in bits, but it can also be set/get in bytes. If the bit length is not a multiple of eight, the `getByteLength()` method will round it up.

```
void setBitLength(int64_t l);
void setByteLength(int64_t l);
void addBitLength(int64_t delta);
void addByteLength(int64_t delta);
int64_t getBitLength() const;
int64_t getByteLength() const;
```

Another extra field is the bit error flag. It can be accessed with the following methods:

```
void setBitError(bool e);
bool hasBitError() const;
```

5.4.2 Identifying the Protocol

In OMNEST protocol models, the protocol type is usually represented in the message subclass. For example, instances of class `IPv6Datagram` represent IPv6 datagrams and `EthernetFrame`

represents Ethernet frames. The C++ `dynamic_cast` operator can be used to determine if a message object is of a specific protocol.

An example:

```
cMessage *msg = receive();
if (dynamic_cast<IPv6Datagram *>(msg) != nullptr) {
    IPv6Datagram *datagram = (IPv6Datagram *)msg;
    ...
}
```

5.4.3 Information About the Last Transmission

When a packet has been received, some information can be obtained about the transmission, namely the *transmission duration* and the *is-reception-start* flag. They are returned by the following methods:

```
simtime_t getDuration() const;
bool isReceptionStart() const;
```

5.4.4 Encapsulating Packets

When modeling layered protocols of computer networks, it is commonly needed to encapsulate a packet into another. The following `cPacket` methods are associated with encapsulation:

```
void encapsulate(cPacket *packet);
cPacket *decapsulate();
cPacket *getEncapsulatedPacket() const;
```

The `encapsulate()` function encapsulates a packet into another one. The length of the packet will grow by the length of the encapsulated packet. An exception: when the encapsulating (outer) packet has zero length, OMNEST assumes it is not a real packet but an out-of-band signal, so its length is left at zero.

A packet can only hold one encapsulated packet at a time; the second `encapsulate()` call will result in an error. It is also an error if the packet to be encapsulated is not owned by the module.

Decapsulation, that is, removing the encapsulated packet, is done by the `decapsulate()` method. `decapsulate()` will decrease the length of the packet accordingly, except if it was zero. If the length would become negative, an error occurs.

The `getEncapsulatedPacket()` function returns a pointer to the encapsulated packet, or `nullptr` if no packet is encapsulated.

Example usage:

```
cPacket *data = new cPacket("data");
data->setByteLength(1024);

UDPPacket *udp = new UDPPacket("udp"); // subclassed from cPacket
udp->setByteLength(8);

udp->encapsulate(data);
EV << udp->getByteLength() << endl; // --> 8+1024 = 1032
```

And the corresponding decapsulation code:

```
cPacket *payload = udp->decapsulate();
```

5.4.5 Reference Counting

Since the 3.2 release, OMNEST implements reference counting of encapsulated packets, meaning that when a packet containing an encapsulated packet is cloned (`dup()`), the encapsulated packet will not be duplicated, only a reference count is incremented. Duplication of the encapsulated packet is deferred until `decapsulate()` actually gets called. If the outer packet is deleted without its `decapsulate()` method ever being called, then the reference count of the encapsulated packet is simply decremented. The encapsulated packet is deleted when its reference count reaches zero.

Reference counting can significantly improve performance, especially in LAN and wireless scenarios. For example, in the simulation of a broadcast LAN or WLAN, the IP, TCP and higher layer packets won't be duplicated (and then discarded without being used) if the MAC address doesn't match in the first place.

The reference counting mechanism works transparently. However, there is one implication: **one must not change anything in a packet that is encapsulated into another!** That is, `getEncapsulatedPacket()` should be viewed as if it returned a pointer to a read-only object (it returns a `const` pointer indeed), for quite obvious reasons: the encapsulated packet may be shared between several packets, and any change would affect those other packets as well.

5.4.6 Encapsulating Several Packets

The `cPacket` class does not directly support encapsulating more than one packet, but one can subclass `cPacket` or `cMessage` to add the necessary functionality.

Encapsulated packets can be stored in a fixed-size or a dynamically allocated array, or in a standard container like `std::vector`. In addition to storage, object ownership needs to be taken care of as well. The message class has to **take ownership** of the inserted messages, and **release** them when they are removed from the message. These tasks are done via the `take()` and `drop()` methods.

Here is an example that assumes that the class has an `std::list` member called `messages` for storing message pointers:

```
void MultiMessage::insertMessage(cMessage *msg)
{
    take(msg);  // take ownership
    messages.push_back(msg);  // store pointer
}

void MultiMessage::removeMessage(cMessage *msg)
{
    messages.remove(msg);  // remove pointer
    drop(msg);  // release ownership
}
```

One also needs to provide an `operator=()` method to make sure that message objects are copied and duplicated properly. Section 7.13 covers requirements and conventions associated with deriving new classes in more detail.

5.5 Attaching Objects To a Message

When parameters or objects need to be added to a message, the preferred way to do that is via message definitions, described in chapter 6.

5.5.1 Attaching Objects

The `cMessage` class has an internal `cArray` object which can carry objects. Only objects that are derived from `cObject` can be attached. The `addObject()`, `getObject()`, `hasObject()`, `removeObject()` methods use the object's name (as returned by the `getName()` method) as the key to the array.

An example where the sender attaches an object, and the receiver checks for the object's existence and obtains a pointer to it:

```
// sender:
cHistogram *histogram = new cHistogram("histogram");
msg->addObject(histogram);

// receiver:
if (msg->hasObject("histogram")) {
   cObject *obj = msg->getObject("histogram");
   cHistogram *histogram = check_and_cast<cHistogram *>(obj);
   ...
}
```

One needs to take care that names of the attached objects don't conflict with each other. Note that message parameters (`cMsgPar`, see next section) are also attached the same way, so their names also count.

When no objects are attached to a message (and `getParList()` is not invoked), the internal `cArray` object is not created. This saves both storage and execution time.

Non-`cObject` data can be attached to messages by wrapping them into `cObject`, for example into `cMsgPar` which has been designed expressly for this purpose. `cMsgPar` will be covered in the next section.

5.5.2 Attaching Parameters

The preferred way of extending messages with new data fields is to use message definitions (see chapter 6).

The old, deprecated way of adding new fields to messages is via attaching `cMsgPar` objects. There are several downsides of this approach, the worst being large memory and execution time overhead. `cMsgPar`'s are heavy-weight and fairly complex objects themselves. It has been reported that using `cMsgPar` message parameters might account for a large part of execution time, sometimes as much as 80%. Using `cMsgPar` is also error-prone because `cMsgPar` objects have to be added dynamically and individually to each message object. In contrast, subclassing benefits from static type checking: if one mistypes the name of a field in the C++ code, the compiler can detect the mistake.

If one still needs `cMsgPar`s for some reason, here is a short summary. At the sender side, one can add a new named parameter to the message with the `addPar()` member function, then set its value with one of the methods `setBoolValue()`, `setLongValue()`, `setStringValue()`,

`setDoubleValue()`, `setPointerValue()`, `setObjectValue()`, and `setXMLValue()`. There are also overloaded assignment operators for the corresponding C/C++ types.

At the receiver side, one can look up the parameter object on the message by name and obtain a reference to it with the `par()` member function. `hasPar()` can be used to check first whether the message object has a parameter object with the given name. Then the value can be read with the methods `boolValue()`, `longValue()`, `stringValue()`, `doubleValue()`, `pointerValue()`, `objectValue()`, `xmlValue()`, or by using the provided overloaded type cast operators.

Example usage:

```
msg->addPar("destAddr");
msg->par("destAddr").setLongValue(168);
...
long destAddr = msg->par("destAddr").longValue();
```

Or, using overloaded operators:

```
msg->addPar("destAddr");
msg->par("destAddr") = 168;
...
long destAddr = msg->par("destAddr");
```

Chapter 6

Message Definitions

6.1 Introduction

In practice, one needs to add various fields to `cMessage` or `cPacket` to make them useful. For example, when modeling communication networks, message/packet objects need to carry protocol header fields. Since the simulation library is written in C++, the natural way of extending `cMessage`/`cPacket` is via subclassing them. However, at least three items has to be added to the new class for each field (a private data member, a getter and a setter method) and the resulting class needs to integrate with the simulation framework, which means that writing the necessary C++ code can be a tedious and time-consuming task.

OMNEST offers a more convenient way called *message definitions*. Message definitions offer a compact syntax to describe message contents, and the corresponding C++ code is automatically generated from the definitions. When needed, the generated class can also be customized via subclassing. Even when the generated class needs to be heavily customized, message definitions can still save the programmer a great deal of manual work.

6.1.1 The First Message Class

Let us begin with a simple example. Suppose that we need a packet type that carries a source and a destination address as well as a hop count. The corresponding C++ code can be generated from the following definition in a `MyPacket.msg` file:

```
packet MyPacket
{
    int srcAddress;
    int destAddress;
    int remainingHops = 32;
};
```

It is the task of the OMNEST *message compiler*, `opp_msgc` or `opp_msgtool`, to translate the definition into a C++ class that can be instantiated from C++ model code. The message compiler is normally invoked for `.msg` files automatically, as part of the build process.

When the message compiler processes `MyPacket.msg`, it creates two files: `MyPacket_m.h` and `MyPacket_m.cc`. The generated `MyPacket_m.h` will contain the following class declaration (abbreviated):

```
class MyPacket : public cPacket {
  protected:
    int srcAddress;
    int destAddress;
    int remainingHops = 32;
  public:
    MyPacket(const char *name=nullptr, short kind=0);
    MyPacket(const MyPacket& other);
    MyPacket& operator=(const MyPacket& other);
    virtual MyPacket *dup() const override {return new MyPacket(*this);}
    ...

    // field getter/setter methods
    virtual int getSrcAddress() const;
    virtual void setSrcAddress(int srcAddress);
    virtual int getDestAddress() const;
    virtual void setDestAddress(int destAddress);
    virtual int getRemainingHops() const;
    virtual void setRemainingHops(int remainingHops);
};
```

As you can see, for each field the generated class contains a protected data member, and a public getter and a setter method. The names of the methods will begin with `get` and `set`, followed by the field name with its first letter converted to uppercase.

The `MyPacket_m.cc` file contains implementation of the generated `MyPacket` class as well as "reflection" code (see `cClassDescriptor`) that allows inspection of these data structures under graphical user interfaces like Qtenv. The `MyPacket_m.cc` file should be compiled and linked into the simulation; this is normally taken care of automatically.

In order to use the `MyPacket` class from a C++ source file, the generated header file needs to be included:

```
#include "MyPacket_m.h"

...
MyPacket *pkt = new MyPacket("pkt");
pkt->setSrcAddress(localAddr);
...
```

6.1.2 Ingredients of Message Files

Message files contain the following ingredients:

- *Packet, message, and class definitions* translate into C++ class definitions. The three types are very similar, they practically only differ in the choice of the default base class (`cPacket`, `cMessage`, and no base class, respectively).
- *Struct definitions* translate into C-like structs, where fields are represented with public data members (there are no getters and setters).
- *Enum definitions* translate into C++ enums.
- *Namespace declarations* define the namespace for subsequent definitions.

- *Imports* allow reusing definitions from other `.msg` files.
- *Properties* are metadata annotations of the syntax `@name` or `@name(...)` that may occur on file, class (packet, struct, etc.) definition, and field level as well. There are many predefined properties, and a large subset of them deal with the details of what C++ code to generate for the item they occur with. For example, `@getter(getFoo)` on a field requests that the generated getter function have the name `getFoo`.
- *C++ blocks* are used for injecting literal C++ code fragments into the generated source files. The target (the place where to insert the code) can be specified.

The following sections describe all of the above elements in detail.

6.2 Classes, Messages, Packets, Structs

As shown above, the message description language allows you to generate C++ data classes and structs from concise descriptions that have a syntax resembling C structs. The descriptions contain the choice of the base class (message descriptions only support single inheritance), the list of fields the class should have, and possibly various metadata annotations that e.g. control the details of the code generation.

A description starts with one of the **`packet`**, **`message`**, **`class`**, **`struct`** keywords. The first three are very similar: they all generate C++ classes, and only differ on the choice of the default base class (and related details such as the argument list of the constructor). The fourth one generates a plain (C-style) struct.

6.2.1 Classes, Messages, Packets

For **`packet`**, the default base class is `cPacket`; or if a base class is explicitly named, it must be a subclass of `cPacket`. Similarly, for **`message`**, the default base class is `cMessage`, or if a base class is specified, it must be a subclass of `cMessage`.

For **`class`**, the default is *no* base class. However, it is often a good idea to choose `cObject` as a base class.[1]

> **NOTE:** It is recommended to use `cObject` as base class, because it adds zero overhead to the generated class, and at the same time makes the class more interoperable with the rest of the simulation library. `cObject` only defines virtual methods but no data members, so the only overhead would be the *vptr*; however, the generated class already has a vptr because the generated methods are also virtual.

The base class is specified with the **`extends`** keyword. For example:

```
packet FooPacket extends PacketBase
{
    ...
};
```

The generated C++ class will look like this:

[1] Until OMNeT++6.0, the default base class was `cObject`. Thus, when migrating code from version 5.x or earlier, one needs to add `extends cObject` to class definitions lacking an "extends" clause.

```
class FooPacket : public PacketBase {
    ...
};
```

The generated class will have a constructor and also a copy constructor. An assignment operator (`operator=()`) and cloning method (`dup()`) will also be generated.

The argument list of the generated constructor depends on the base class. For classes derived from `cMessage`, it will accept an object name and message kind. For classes derived from `cNamedObject`, it will accept an object name. The arguments are optional (they have default values).

```
class FooPacket : public PacketBase
{
  public:
    FooPacket(const char *name=nullptr, int kind=0);
    FooPacket(const FooPacket& other);
    FooPacket& operator=(const FooPacket& other);
    virtual FooPacket *dup() const;
    ...
```

Additional base classes can be added by listing them in the `@implements` class property.

6.2.2 Structs

Message definitions allow one to define C-style structs, "C-style" meaning "containing only data and no methods". These structs can be useful as fields in message classes.

The syntax is similar to that of defining messages:

```
struct Place
{
    int type;
    string description;
    double coords[3];
};
```

The generated struct has public data members, and no getter or setter methods. The following code is generated from the above definition:

```
// generated C++
struct Place
{
    int type;
    omnetpp::opp_string description;
    double coords[3];
};
```

Note that **`string`** fields are generated with the `opp_string` C++ type, which is a minimalistic string class that wraps `const char*` and takes care of allocation/deallocation. It was chosen instead of `std::string` because of its significantly smaller memory footprint. (`std::string` is significantly larger than a `const char*` pointer because it also needs to store length and capacity information in some form.)

Inheritance is supported for structs:

```
struct Base
{
    ...
};

struct Extended extends Base
{
    ...
};
```

However, because a struct has no member functions, there are limitations:

- variable-size arrays are not supported;
- customization via inheritance and **abstract** fields (see later in 6.10.6) cannot be used;
- cannot have classes subclassed from `cOwnedObject` as fields, because structs cannot be owners.

6.3 Enums

An enum is declared with the **enum** keyword, using the following syntax:

```
enum PayloadType
{
   NONE = 0;
   VOICE = 1;
   VIDEO = 2;
   DATA = 3;
};
```

Enum values need to be unique.

The message compiler translates an enum into a normal C++ enum, plus also generates a descriptor that stores the symbolic names as strings. The latter makes it possible for Qtenv to display symbolic names for enum values.

Enums can be used in two ways. The first is simply to use the enum's name as field type:

```
packet FooPacket
{
    PayloadType payloadType;
};
```

The second way is to tag a field of the type **int** or any other integral type with the `@enum` property and the name of the enum, like so:

```
packet FooPacket
{
    int16_t payloadType @enum(PayloadType);
};
```

In the generated C++ code, the field will have the original type (in this case, **int16_t**). However, additional code generated by the message compiler will allow Qtenv to display the symbolic name of the field's value in addition to the numeric value.

6.4 Imports

Import directives are used to make definitions in one message file available to another one. Importing an MSG file makes the definitions in that file available to the file that imports it, but has no further side effect (and in particular, it will generate no C++ code).

To import a message file, use the **import** keyword followed by a name that identifies the message file within its project:

```
import inet.linklayer.common.MacAddress;
```

The **import**'s parameter is interpreted as a relative file path (by replacing dots with slashes, and appending `.msg`), which is searched for in folders listed in the *message import path*, much like C/C++ include files are searched for in the compiler's include path, Python modules in the Python module search path, or NED files in the NED path.

The message import path can be be specified to the message compiler via a series of `-I` command-line options.

6.5 Namespaces

To place generated types into a namespace, add a **namespace** directive above the types in question:

```
namespace inet;
```

Hierarchical (nested) namespaces are declared using double colons in the namespace definition, much like nested namespace definitions introduced into C++ in version C++17.

```
namespace inet::ieee80211;
```

The above code will be translated into multiple nested namespaces in the C++ code:

```
namespace inet { namespace ieee80211 {
...
}}
```

There can be multiple **namespace** directives in a message file. The effect of the **namespace** directive extends from the place of the directive until the next **namespace** directive or the end of the message file. Each **namespace** directive opens a completely new namespace, i.e. *not* a namespace within the previous one. An empty namespace directive (`namespace;`) returns to the global namespace. For example:

```
namespace foo::bar;
class A {}  // defines foo::bar::A

namespace baz;
class B {}  // defines baz::B

namespace;
class C {}  // defines ::C
```

6.6 Properties

Properties are metadata annotations of the syntax `@name` or `@name(...)` that may occur on file, class (packet, struct, etc.) definition, and field level. There are many predefined properties, and a large subset of them deal with the details of what C++ code to generate for the item they occur with. For example, `@getter(getFoo)` on a field requests that the generated getter function have the name `getFoo`.

Here is a syntax example. Note that class properties are placed in the fields list (fields and properties may be mixed in arbitrary order), and field properties are written after the field name.

```
@foo;
class Foo {
  @customize(true);
  string value @getter(...) @setter(...) @hint("...");
}
```

Syntactically, the mandatory part of a property is the `@` character followed by the property name. They are then optionally followed by an *index* and a *parameter list*. The index is a name in square brackets, and it is rarely used. The parameter list is enclosed in parentheses, and in theory it may contain a value list and key-valuelist pairs, but almost all properties expect to find just a single value there.

For boolean properties, the value may be **`true`** or **`false`**; if the value is missing, **`true`** is assumed. Thus, `@customize` is equivalent to `@customize(true)`.

As a guard against mistyping property names, properties need to be declared before they can be used. Properties are declared using the `@property` property, with the name of the new property in the index, and the type and other attributes of the property in the parameter list. Examples for property declarations, including the declaration of `@property` itself, can be seed by listing the built-in definitions of the message compiler (`opp_msgtool -h builtindefs`).

The full list of properties understood by the message compiler and other OMNEST tools can be found in Appendix F.

6.6.1 Data Types

The following data types can be used for fields:

- C/C++ primitive data types: **`bool`**, **`char`**, **`short`**, **`int`**, **`long`**, **`unsigned char`**, **`unsigned short`**, **`unsigned int`**, **`unsigned long`**, **`float`**, **`double`**.
- **`string`**. Getters and setters use the **`const char*`** data type; **`nullptr`** is not allowed. Setters store a copy of the string, not just the pointer.
- C99-style fixed-size integer types: **`int8_t`**, **`int16_t`**, **`int32_t`**, **`int64_t`**, **`uint8_t`**, **`uint16_t`**, **`uint32_t`**, **`uint64_t`**.[2]

In addition, OMNEST class names such as `simtime_t` and `cMessage` are also made available without the need to import anything. These names are accepted both with and without spelling out the `omnetpp` namespace name.

Numeric fields are initialized to zero, booleans to **`false`**, and string fields to the empty string.

[2] These type names are accepted without the `_t` suffix as well, but you are responsible to ensure that the generated code compiles, i.e. the shortened type names must be defined in a header file you include.

6.7 Fields

6.7.1 Scalar fields

A scalar field is one that holds a single value. It is defined by specifying the data type and the field name, for example:

```
int timeToLive;
```

For each field, the generated class will have a protected data member, and a public getter and setter method. The names of the methods will begin with `get` and `set`, followed by the field name with its first letter converted to uppercase. Thus, the above field will generate the following methods in the C++ class:

```
int getTimeToLive() const;
void setTimeToLive(int timeToLive);
```

> **NOTE:** All methods are generated to be virtual, but we omit the **`virtual`** keyword here and in further examples.

The method names are derived from the field name, but they can be customized with the `@getter` and `@setter` properties, as shown below:

```
int timeToLive @getter(getTTL) @setter(setTTL);
```

The choice of C++ type used for the data member and the getter/setter methods can be overridden with the help of the `@cppType` property (and on a more fine-grained level, with `@datamemberType`, `@argType` and `@returnType`), although this it is rarely useful.

6.7.2 Initial Values

Initial values for fields can be specified after an equal sign, like so:

```
int version = HTTP_VERSION;
string method = "GET";
string resource = "/";
bool keepAlive = true;
int timeout = 5*60;
```

Any phrase that is a valid C++ expression can be used as initializer value. (The message compiler does not check the syntax of the values, it merely copies them into the generated C++ file.)

For array fields, the initializer specifies the value for individual array elements. There is no syntax for initializing an array with a list of values.

6.7.3 Overriding Initial Values from Subclasses

In a subclass, it is possible to override the initial value of an inherited field. The syntax is similar to that of a field definition with initial value, only the data type is missing.

An example:

```
packet Ieee80211Frame
{
    int frameType;
    ...
};

packet Ieee80211DataFrame extends Ieee80211Frame
{
    frameType = DATA_FRAME;  // assignment of inherited field
    ...
};
```

It may seem like the message compiler would need the definition of the base class to check the definition of the field being assigned. However, it is not the case. The message compiler trusts that such field exists; or rather, it leaves the check to the C++ compiler.

What the message compiler actually does is derives a setter method name from the field name, and generates a call to it into the constructor. Thus, the generated constructor for the above packet type would be something like this:

```
Ieee80211DataFrame::Ieee80211DataFrame(const char *name, int kind) :
    Ieee80211Frame(name, kind)
{
    this->setFrameType(DATA_FRAME);
    ...
}
```

This implementation also lets one initialize `cMessage` / `cPacket` fields such as message kind or packet length:

```
packet UDPPacket
{
    byteLength = 16;  // results in 'setByteLength(16);' being placed into ctor
};
```

6.7.4 Const Fields

A field can be marked as const by the using **`const`** keyword. A const field only has a (const) data member and a getter function, but no setter. The value can be provided via an initializer. An example:

```
const int foo = 24;
```

This generates a `const int` data member in the class, initialized to 24, and a getter member function that returns its value:

```
int getFoo() const;
```

Array fields cannot be const.

Note that a pointer field may also be marked const, but **`const`** is interpreted differently in that case: as a mutable field that holds a pointer to a **`const`** object.

One use of **`const`** is to implement computed fields. For that, the field needs to be annotated with the `@custom` or `@customImpl` property to allow for a custom implementation to be sup-

plied for the getter. The custom getter can then encapsulate the computation of the field value. Customization is covered in section 6.10.

> **NOTE:** To add actual constants (as opposed to getter-only fields) to a class, it is better use a targeted **cplusplus** block to inject their definitions into the C++ class declaration.

6.7.5 Abstract Fields

Abstract fields is a way to allow a custom implementation (such as storage, getter/setter methods, etc.) to be provided for a field. For a field marked as abstract, the message compiler does not generate a data member, and generated getter/setter methods will be pure virtual. It is expected that the pure virtual methods will be implemented in a subclass (possibly via `@customize`, see section 6.10).

A field is declared abstract by using the **abstract** keyword or the `@abstract` property (the two are equivalent).

```
abstract bool urgentBit; // or: bool urgentBit @abstract;
```

The generated pure virtual methods:

```
virtual bool getUrgentBit() const = 0;
virtual void setUrgentBit(bool urgentBit) = 0;
```

Alternatives to **abstract**, at least for certain use cases, are `@custom` and `@customImpl` (see section 6.10).

6.7.6 Fixed-Size Arrays

Fixed-size arrays can be declared with the usual syntax of putting the array size in square brackets after the field name:

```
int route[4];
```

The generated getter and setter methods will have an extra `k` argument (the array index), and a third method that returns the array size is also generated:

```
int getRoute(size_t k) const;
void setRoute(size_t k, int route);
size_t getRouteArraySize() const;
```

When the getter or setter method is called with an index that is out of bounds, an exception is thrown.

The method names can be overridden with the `@getter`, `@setter` and `@sizeGetter` properties. To use another C++ type for array size and indices instead of the default `size_t`, specify the `@sizeType` property.

> **NOTE:** Use a singular noun for field name instead of plural (`route[]` instead of `routes[]`), otherwise method names will look confusing (`getRoutes()`, `appendRoutes()`, etc, for methods that deal with a single route).

When a default value is given, it is interpreted as a scalar for filling the array with. There is no syntax for initializing an array with a list of values.

```
int route[4] = -1; // all elements set to -1
```

6.7.7 Variable-Size Arrays

If the array size is not known in advance, the field can be declared to have a variable size by using an empty pair in brackets:

```
int route[];
```

In this case, the generated class will have extra methods in addition to the getter and setter: one for resizing the array, one for getting the array size, plus methods for inserting an element at a given position, appending an element, and erasing an element at a given position.

```
int getRoute(size_t k) const;
void setRoute(size_t k, int route);
void setRouteArraySize(size_t size);
size_t getRouteArraySize() const;
void insertRoute(size_t k, int route);
void appendRoute(int route);
void eraseRoute(size_t k);
```

The default array size is zero. Elements can be added by calling the inserter or the appender method, or resizing the array and setting individual elements.

Internally, all methods that change the array size (inserter, appender, resizer) always allocate a new array, and copy existing values over to the new array. Therefore, when adding a large number elements, it is recommended to resize the array first, instead of calling the appender method multiple times.

The method names can be overridden with the `@getter`, `@setter`, `@sizeGetter`, `@sizeSetter`, `@inserter`, `@appender` and `@eraser` field properties. To use another C++ type for array size and indices instead of the default `size_t`, specify the `@sizeType` property.

When a default value is given, it is used for initializing new elements when the array is expanded.

```
int route[] = -1;
```

6.7.8 Classes and Structs as Fields

Classes and structs may also be used as as fields, not only primitive types and `string`. For example, given a class named `IPAddress`, one can write the following field:

```
IPAddress sourceAddress;
```

The `IPAddress` type must be known to the message compiler.

The generated class will contain an `IPAddress` data member, and the following member functions:

```
const IPAddress& getSourceAddress() const;
void setSourceAddress(const IPAddress& sourceAddress);
IPAddress& getSourceAddressForUpdate();
```

Note that in addition to the getter and setter, a mutable getter (`get...ForUpdate`) is also generated, which allows the stored value (object or struct) to be modified in place.

By default, values are passed by reference. This can be changed by specifying the `@byValue` property:

```
IPAddress sourceAddress @byValue;
```

This generates the following member functions:

```
virtual IPAddress getSourceAddress() const;
virtual void setSourceAddress(IPAddress sourceAddress);
```

Note that both member functions use pass-by-value, and that the mutable getter function is not generated.

Specifying **const** will cause only a getter function to be generated but no setter or mutable getter, as shown before in 6.7.4.

Array fields are treated similarly, the difference being that the getter and setter methods take an extra index argument:

```
IPAddress route[];
```

The generated methods:

```
void setRouteArraySize(size_t size);
size_t getRouteArraySize() const;
const IPAddress& getRoute(size_t k) const;
IPAddress& getRouteForUpdate(size_t k);
void setRoute(size_t k, const IPAddress& route);
void insertRoute(size_t k, const IPAddress& route);
void appendRoute(const IPAddress& route);
void eraseRoute(size_t k);
```

6.7.9 Non-Owning Pointer Fields

The field type may be a pointer, both for scalar and array fields. Pointer fields come in two flavours: owning and non-owning. A non-owning pointer field just stores the pointer value regardless of the ownership of the object it points to, while an owning pointer holds the ownership of the object. This section discusses non-owning pointer fields.

Example:

```
cModule *contextModule;  // missing @owner: non-owning pointer field
```

The generated methods:

```
const cModule *getContextModule() const;
void setContextModule(cModule *contextModule);
cModule *getContextModuleForUpdate();
```

If the field is marked **const**, then the setter will take a **const** pointer, and the `getForUpdate()` method is not generated:

```
const cModule *contextModule;
```

The output:

```
const cModule *getContextModule() const;
void setContextModule(const cModule *contextModule);
```

6.7.10 Owning Pointer Fields

This section discusses pointer fields that own the objects they point to, that is, are responsible for deallocating the object when the object containing the field (let's refer to it as *container* object) is deleted.

For all owning pointer fields in a class, the destructor of the class deletes the owned objects, the `dup()` method and the copy constructor duplicate the owned objects for the newly created object, and the assignment operator (`operator=`) does both: the old objects in the destination object are deleted, and replaced by clones of the objects in the source object.

When the owned object is a subclass of `cOwnedObject` that keeps track of its owner, the code generated for the container class invokes the `take()` and `drop()` methods at the appropriate times to manage the ownership.

Example:

```
cPacket *payload @owned;
```

The generated methods:

```
const cPacket *getPayload() const;
cPacket *getPayloadForUpdate();
void setPayload(cPacket *payload);
cPacket *removePayload();
```

The getter and mutable getter return the stored pointer (or **`nullptr`** if there is none).

The remover method releases the ownership of the stored object, sets the field to **`nullptr`**, and returns the object.

The setter method behavior depends on the presence of the `@allowReplace` property. By default (when `@allowReplace` is absent), the setter does not allow replacing the object. That is, when the setter is invoked on a field that already contains an object (the pointer is non-null), an error is raised: *"A value is already set, remove it first with removePayload()"*. One must call `removePayload()` before setting a new object.

When `@allowReplace` is specified for the field, there is no need to call te remover method before setting a new value, because the setter method deletes the old object before storing the new one.

```
cPacket *payload @owned @allowReplace; // allow setter to delete the old object
```

If the field is marked **`const`**, then the `getForUpdate()` method is not generated, and the setter takes a **`const`** pointer.

```
const cPacket *payload @owned;
```

The generated methods:

```
const cPacket *getPayload() const;
void setPayload(const cPacket *payload);
cPacket *removePayload();
```

The name of the remover method (which is the only extra method compared to non-pointer fields) can be customized using the `@remover` property.

6.8 Literal C++ Blocks

It is possible to have C++ code fragments injected directly into the generated code. This is done with the **cplusplus** keyword optionally followed by a *target* in parentheses, and the code fragment enclosed in double curly braces.

The target specifies where to insert the code fragment in the generated header or implementation file; we'll get to it in a minute.

As far as a the code fragment is concerned, the message compiler does not try to make sense of it, just simply copies it into the generated source file at the requested location. The code fragment should be formatted so that it does not contain a double close curly brace (`}}`) because it would be interpreted as end of the fragment block.[3]

```
cplusplus {{
#include "FooDefs.h"
#define SOME_CONSTANT 63
}}
```

The target can be `h` (the generated header file – this is the default), `cc` (the generated `.cc` file), the name of a type generated in the same message file (content is inserted in the declaration of the type, just before the closing curly brace), or a member function name of one such type.

cplusplus blocks with the target `h` are customarily used to insert `#include` directives, commonly used constants or macros (e.g. `#define`s), or, rarely, typedefs and other elements into the generated header. The fragments are pasted into the namespace which is open at that point. Note that includes should always be placed into a **cplusplus(h)** block **above** the first namespace declaration in the message file.

cplusplus blocks with `cc` as target allow you to insert code into the `.cc` file, e.g. implementations of member functions. This is useful e.g with custom-implementation fields (`@customImpl`, see 6.10.4).

cplusplus blocks with a type name as target allow you to insert new data members and member functions into the class. This is useful e.g with custom fields (`@custom`, see 6.10.5).

To inject code into the implementation of a member function of a generated class, specify <*classname*>::<*methodname*> as target. Supported methods include the constructor, copy constructor (use `Foo&` as name), destructor, `operator=`, `copy()`, `parsimPack()`, `parsimUnpack()`, etc., and the per-field generated methods (setter, getter, etc.).

6.9 Using External C++ Types

The message compiler only allows types it knows about to be used for fields or base classes. If you want to use to types not generated by the message compiler, you need to do the following:

1. Let the message compiler know about the type; and
2. Make sure its C++ declaration is available at compile time

For the first one can be achieved with the `@existingClass` property. When a type (class or struct) is annotated with `@existingClass`, the message compiler remembers the definition,

[3] Should this ever be a problem, just insert a space between the two braces, or use the automatic concatenation of adjacent string literals feature of C/C++ if they occur within a string constant, i.e. break up `"foo}}bar"` into `"foo}" "}bar"`.

but assumes that the class (or struct) already exist in C++ code, and does not generate it. (However, it will still generate a class descriptor, see section 6.11.)

> **NOTE:** Support for C++-style type announcements is no longer part of the message definitions syntax, they were removed in OMNEST version 6.0.

The second task is achieved by adding a **cplusplus** block with an `#include` directive to the message file.

For example, suppose we have a hand-written `ieee802::MACAddress` class defined in `MACAddress.h` that we would like to use for fields in multiple message files. One way to make this possible is to add a `MACAddress.msg` file alongside the header with the following content:

```
// MACAddress.msg

cplusplus {{
#include "MACAddress.h"
}}

class ieee802::MACAddress  // a separate namespace decl would also do
{
    @existingClass;
    int8_t octet[6]; // assumes class has getOctet(k) and setOctet(k)
}
```

As exemplified above, for existing classes it is possible to announce them with their namespace-qualified name, there is no need for separate **namespace** line.

This message file can be imported into all other message files that need `MACAddress`, for example like this:

```
import MACAddress;

packet EthernetFrame {
    ieee802::MACAddress source;
    ieee802::MACAddress destination;
    ...
}
```

6.10 Customizing the Generated Class

There are several possibilities for customizing a generated class:

- Using custom method names and custom field types
- Using custom field types
- Injecting code into existing member functions
- Custom fields
- Fields with custom-implementation methods
- The Generation Gap pattern

- Abstract fields
- Special customizations, e.g. `@str`, `@nopack`, etc.

The following sections explore the above possibilities.

6.10.1 Customizing Method Names

The names and some other properties of generated methods can be influenced with metadata annotations (properties).

The following field properties exist for overriding method names: `@getter`, `@setter`, `@getterForUpdate`, `@remover`, `@sizeGetter`, `@sizeSetter`, `@inserter`, `@appender` and `@eraser`.

To override data types used by the data member and its accessor methods, use `@cppType`, `@datamemberType`, `@argType`, or `@returnType`.

To override the default `size_t` type used for array size and indices, use `@sizeType`.

Consider the following example:

```
packet IPPacket {
    int ttl @getter(getTTL) @setter(setTTL);
    Option options[] @sizeGetter(getNumOptions)
                     @sizeSetter(setNumOptions)
                     @sizetype(short);
}
```

The generated class would have the following methods (note the differences from the default names `getTtl()`, `setTtl()`, `getOptions()`, `setOptions()`, `getOptionsArraySize()`, `getOptionsArraySize()`; also note that indices and array sizes are now **`short`**):

```
virtual int getTTL() const;
virtual void setTTL(int ttl);
virtual const Option& getOption(short k) const;
virtual void setOption(short k, const Option& option);
virtual short getNumOptions() const;
virtual void setNumOptions(short n);
```

In some older simulation models you may also see the use of the `@omitGetVerb` class property. This property tells the message compiler to generate getter methods without the "get" prefix, e.g. for a `sourceAddress` field it would generate a `sourceAddress()` method instead of the default `getSourceAddress()`. It is not recommended to use `@omitGetVerb` in new models, because it is inconsistent with the accepted naming convention.

6.10.2 Injecting Code into Methods

Generally, literal C++ blocks (the **`cplusplus`** keyword) are the way to inject code into the body of individual methods, as described in 6.8.

The `@beforeChange` class property can be used to designate a member function which is to be called before any mutator code (in setters, non-const getters, assignment operator, etc.) executes. This can be used to implement e.g. a dirty flag or some form of immutability (i.e. freeze the state of the object).

6.10.3 Generating str()

The `@str` class property aims at simplifying adding an `str()` method in the generated class. Having an `str()` method is often useful for debugging, and it also has a special role in class descriptors (see 6.11.6).

When `@str` is present, an `std::string str() const` method is generated for the class. The method's implementation will contain a single `return` keyword, with the value of the `@str` property copied after it.

Example:

```
class Location {
    double lat;
    double lon;
    @str("(" + std::to_string(getLat()) + "," + std::to_string(getLon()) + ")");
}
```

It will result in the following `str()` method to be generated as part of the `Location` class:

```
std::string Location::str() const
{
    return "(" + std::to_string(getLat()) + "," + std::to_string(getLon()) + ")";
}
```

6.10.4 Custom-implementation Methods

When member functions generated for a field need customized implementation and method-targeted C++ blocks are not sufficient, the `customImpl` property can be of help. When a field is marked `customImpl`, the message compiler will skip generating the implementations of its accessor methods in the `.cc` file, allowing the user to supply their own versions.

Here is a simple example. The methods in it do not perform anything extra compared to the default generated versions, but they illustrate the principle.

```
class Packet
{
    int hopCount @customImpl;
}

cplusplus(cc) {{
int Packet::getHopCount() const
{
    return hopCount; // replace/extend with extra code
}

void Packet::setHopCount(int value)
{
    hopCount = value; // replace/extend with extra code
}
}}
```

6.10.5 Custom Fields

If a field is marked with `@custom`, the field will only appear in the class descriptor, but no code is generated for it at all. One can inject the code that implements the field (data member, getter, setter, etc.) via targeted **cplusplus** blocks (6.8). `@custom` is a good way to go when you want the field to have a different underlying storage or different accessor methods than normally generated by the message compiler. (For the latter case, however, be aware that the generated class descriptor assumes the presence of certain accessor methods for the field, although the set of expected methods can be customized to a degree. See 6.11 for details.)

The following example uses `@custom` to implement a field that acts a stack (has `push()` and `pop()` methods), and uses `std::vector` as the underlying data structure.

```
cplusplus {{
#include <vector>
}}

class MPLSHeader
{
    int32_t label[] @custom @sizeGetter(getNumLabels) @sizeSetter(setNumLabels);
}

cplusplus(MPLSHeader) {{
  protected:
    std::vector<int32_t> labels;
  public:
    // expected methods:
    virtual void setNumLabels(size_t size) {labels.resize(size);}
    virtual size_t getNumLabels() const {return labels.size();}
    virtual int32_t getLabel(size_t k) const {return labels.at(k);}
    virtual void setLabel(size_t k, int32_t label) {labels.at(k) = label;}
    // new methods:
    virtual void pushLabel(int32_t label) {labels.push_back(label);}
    virtual int32_t popLabel() {auto l=labels.back();labels.pop_back();return l;}
}}

cplusplus(MPLSHeader::copy) {{
    labels = other.labels;
}}
```

The last C++ block is needed so that the copy constructor and the `operator=` method also copies the new field. (`copy()` is a member function where the common part of the above two are factored out, and the C++ block injects code in there.)

6.10.6 Customizing the Class via Inheritance

Another way of customizing the generated code is by employing what is known as the *Generation Gap* design pattern, proposed by John Vlissides. The idea is that the customization can be done while *subclassing* the generated class, overriding whichever member functions need to be different from their generated versions.

This feature is enabled by adding the `@customize` property on the class. Doing so will cause the message compiler to generate an intermediate class instead of the final one, and the user

will subclass the intermediate class to obtain the real class. The name of the intermediate class is obtained by appending `_Base` to the class name. The subclassing code can be in an entirely different header and `.cc` file from the generated one, so this method does not require the use of **cplusplus** blocks.

Consider the following example:

```
packet FooPacket
{
   @customize(true);
   ...
};
```

The message compiler will generate a `FooPacket_Base` class instead of `FooPacket`. It is then the user's task to subclass `FooPacket_Base` to derive `FooPacket`, while adding extra data members and adding/overriding methods to achieve the goals that motivated the customization.

There is a minimum amount of code you have to write for `FooPacket`, because not everything can be pre-generated as part of `FooPacket_Base` (e.g. constructors cannot be inherited). This minimum code, which usually goes into a header file, is the following:

```
class FooPacket : public FooPacket_Base
{
  private:
    void copy(const FooPacket& other) { ... }
  public:
    FooPacket(const char *s=nullptr, short kind=0) : FooPacket_Base(s,kind) {}
    FooPacket(const FooPacket& other) : FooPacket_Base(other) {copy(other);}
    FooPacket& operator=(const FooPacket& other) {if (this==&other) return *this;
        FooPacket_Base::operator=(other); copy(other); return *this;}
    virtual FooPacket *dup() const override {return new FooPacket(*this);}
};
```

NOTE: The above boilerplate code can be copied out of the generated C++ header, which contains it as a comment.

The generated constructor, copy constructor, `operator=`, `dup()` can be usually be copied verbatim. The only method that needs to be custom code is `copy()`. It is shared by the copy constructor and `operator=`, and should take care of copying the new data members you added as part of `FooPacket`.

In addition to the above, the implementation (`.cc`) file should contain the registration of the new class:

```
Register_Class(FooPacket);
```

6.10.7 Using an Abstract Field

Abstract fields, introduced in 6.7.5, are an alternative to `@custom` (see 6.10.5) for allowing a custom implementation (such as storage, getter/setter methods, etc.) to be provided for a field. For a field marked **abstract**, the message compiler does not generate a data member, and generated getter/setter methods will be pure virtual.

Abstract fields are most often used together with the Generation Gap pattern (see 6.10.6), so that one can immediately supply a custom implementation.

The following example demonstrates the use of abstract fields for creating an array field that uses `std::vector` as underlying implementation:

```
packet FooPacket
{
    @customize(true);
    abstract int foo[]; // impl will use std::vector<int>
}
```

If you compile the above code, in the generated C++ code you will only find abstract methods for `foo`, but no underlying data member or method implementation. You can implement everything as you like. You can then write the following C++ file to implement `foo` with `std::vector` (some details omitted for brevity):

```
#include <vector>
#include "FooPacket_m.h"

class FooPacket : public FooPacket_Base
{
  protected:
    std::vector<int> foo;

  public:
    // constructor and other methods omitted, see below
    ...
    virtual int getFoo(size_t k) {return foo[k];}
    virtual void setFoo(size_t k, int x) {foo[k]=x;}
    virtual void addFoo(int x) {foo.push_back(x);}
    virtual void setFooArraySize(size_t size) {foo.resize(size);}
    virtual size_t getFooArraySize() const {return foo.size();}
};

Register_Class(FooPacket);
```

Some additional boilerplate code is needed so that the class conforms to conventions, and duplication and copying works properly:

```
    FooPacket(const char *name=nullptr, int kind=0) : FooPacket_Base(name,kind) {
    }
    FooPacket(const FooPacket& other) : FooPacket_Base(other.getName()) {
        operator=(other);
    }
    FooPacket& operator=(const FooPacket& other) {
        if (&other==this) return *this;
        FooPacket_Base::operator=(other);
        foo = other.foo;
        return *this;
    }
    virtual FooPacket *dup() {
        return new FooPacket(*this);
    }
```

6.11 Descriptor Classes

For each generated class and struct, the message compiler also generates an associated descriptor class, which class carries "reflection" information about the new class. The descriptor class encapsulates virtually all information that the original message definition contains, and exposes it via member functions. Reflection information allows inspecting object contents down to field level in Qtenv, filtering objects by a filter expression that refers to object fields, serializing messages-packets in a readable form for the eventlog file, and has several further potential uses.

6.11.1 cClassDescriptor

The descriptor class is subclassed from `cClassDescriptor`. It has methods for enumerating fields (`getFieldCount()`, `getFieldName()`, `getFieldTypeString()`, etc.), for getting and setting a field's value in string form (`getFieldAsString()`, `setFieldAsString()`) and as `cValue` (`getFieldValue()`, `setFieldValue()`), for exploring the class hierarchy (`getBaseClassDescriptor()`, etc.), for accessing class and field properties, and for similar tasks.

Classes derived from `cObject` have a virtual member function `getDescriptor` that returns their associated descriptor. For other classes, it is possible to obtain the descriptor using `cClassDescriptor::getDescriptorFor()` with the class name as argument.

Several properties control the creation and details of the class descriptor.

6.11.2 Controlling Descriptor Generation

The `@descriptor` class property can be used to control the generation of the descriptor class. `@descriptor(readonly)` instructs the message compiler not to generate field setters for the descriptor, and `@descriptor(false)` instructs it not to generate a description class for the class at all.

6.11.3 Generating Descriptors For Existing Classes

It is also possible to use (or abuse) the message compiler for generating a descriptor class for an existing class. To do that, write a message definition for your existing class (for example, if it has `int getFoo()` and `setFoo(int)` methods, add an `int foo` field to the message definition), and mark it with `@existingClass`. This will tell the message compiler that it should not generate an actual class (as it already exists), only a descriptor class.

6.11.4 Field Metadata

When an object is shown in Qtenv's Object Inspector pane, Qtenv obtains all information it displays from the object's descriptor. There are several properties that can be used to customize how a field appears in the Object Inspector:

- `@icon` associates an icon with the field;
- `@label` overrides the text displayed as field name;
- `@group` is used for grouping related fields;

- `@hint` can be used to provide a short description of the field, which Qtenv displays in a tooltip

6.11.5 Method Name Properties

Several of the properties which are for overriding field accessor method names (`@getter`, `@setter`, `@sizeGetter`, `@sizeSetter`, etc., see 6.10.1) have a secondary purpose. When generating a descriptor for an *existing* class (see `@existingClass`), those properties specify how the descriptor can access the field, i.e. what code to generate in the implementation of the descriptor's various methods. In that use case, such properties may contain code fragments or a function call template instead of a method name.

6.11.6 toString/fromString

To be able to generate the descriptor's `getFieldValueAsString()` member function, the message compiler needs to know how to convert the return type of the getter to `std::string`. Similarly, for `setFieldValueAsString()` it needs to know how to convert (or parse) a string to obtain the setter's argument type. For the built-in types (**`int`**, **`double`**, etc.) this information is pre-configured, but for other types the user needs to supply it via two properties:

- `@toString` specifies the code to convert the return type of the setter to a string;
- `@fromString` specifies the code to convert a string to the setter's argument type.

These properties can be specified on the class (where it will be applied to fields of that type), or directly on fields. Multiple syntaxes are accepted:

- If the value starts with a dot, it is interpreted as a member function call.
- If the value contains a dollar sign, it serves as a placeholder for the value to be converted.

Example:

```
class IPAddress
{
    @existingClass;
    @opaque;
    @toString(.str());  // use IPAddress::str() to produce a string
    @fromString(IPAddress($));  // use constructor; '$' will be replaced by the st
}
```

If the `@toString` property is missing, the message compiler generates code which calls the `str()` member function on the value returned by the getter, provided that it knows for certain that the corresponding type has such method (the type is derived from `cObject`, or has the `@str` property).

If there is no `@toString` property and no (known) `str()` method, the descriptor will return the empty string.

6.11.7 toValue/fromValue

Similarly to `@toString`/`@fromString` described in the previous section, the `@toValue` and `@fromValue` properties are used define how to convert the field's value to and from `cValue` for the descriptor's `getFieldValue()` and `setFieldValue()` methods.

6.11.8 Field Modifiers

There are several boolean-valued properties which enable/disable various features in the descriptor:

- `@opaque`: If true: Treat the field as atomic (non-compound) type, i.e. having no descriptor class. When specified on a class, it determines the default for fields of that type.
- `@editable`: If set, value of the field (or value of fields that are instances of this type) can be set via the class descriptor's `setFieldValueFromString()` and `setFieldValue()` methods.
- `@replaceable`: If set, field is a pointer whose value can be set via the class descriptor's `setFieldStructValuePointer()` and `setFieldValue()` methods.
- `@resizable`: If set, field is a variable-size array whose size can be set via the class descriptor's `setFieldArraySize()` method.
- `@readonly`: This is simply a shorthand for `@editable(false) @replaceable(false) @resizable(false)`.

Chapter 7

The Simulation Library

OMNEST has an extensive C++ class library available to the user for implementing simulation models and model components. Part of the class library's functionality has already been covered in the previous chapters, including discrete event simulation basics, the simple module programming model, module parameters and gates, scheduling events, sending and receiving messages, channel operation and programming model, finite state machines, dynamic module creation, signals, and more.

This chapter discusses the rest of the simulation library. Topics will include logging, random number generation, queues, topology discovery and routing support, and statistics and result collection. This chapter also covers some of the conventions and internal mechanisms of the simulation library to allow one extending it and using it to its full potential.

7.1 Fundamentals

7.1.1 Using the Library

Classes in the OMNEST simulation library are part of the `omnetpp` namespace. To use the OMNEST API, one must include the `omnetpp.h` header file and either import the namespace with `using namespace omnetpp`, or qualify names with the `omnetpp::` prefix.

Thus, simulation models will contain the

```
#include <omnetpp.h>
```

line, and often also

```
using namespace omnetpp;
```

When writing code that should work with various versions of OMNEST, it is often useful to have compile-time access to the OMNEST version in a numeric form. The `OMNETPP_VERSION` macro exists for that purpose, and it is defined by OMNEST to hold the version number in the form *major*256+minor*. For example, in OMNEST 4.6 it was defined as

```
#define OMNETPP_VERSION 0x406
```

7.1.2 The cObject Base Class

Most classes in the simulation library are derived from `cObject`, or its subclasses `cNamedObject` and `cOwnedObject`. `cObject` defines several virtual member functions that are either inherited or redefined by subclasses. Otherwise, `cObject` is a zero-overhead class as far as memory consumption goes: it purely defines an interface but has no data members. Thus, having `cObject` a base class does not add anything to the size of a class if it already has at least one virtual member function.

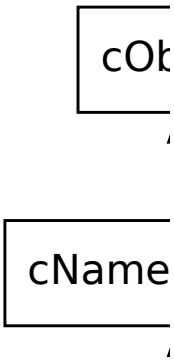


Figure 7.1: cObject is the base class for most of the simulation library

The subclasses `cNamedObject` and `cOwnedObject` add data members to implement more functionality. The following sections discuss some of the practically important functonality defined by `cObject`.

Name and Full Name

The most useful and most visible member functions of `cObject` are `getName()` and `getFullName()`. The idea behind them is that many objects in OMNEST have names by default (for example, modules, parameters and gates), and even for other objects, having a printable name is a huge gain when it comes to logging and debugging.

`getFullName()` is important for gates and modules, which may be part of gate or module vectors. For them, `getFullName()` returns the name with the index in brackets, while `getName()` only returns the name of the module or gate vector. That is, for a gate `out[3]` in the gate vector `out[10]`, `getName()` returns `"out"`, and `getFullName()` returns `"out[3]"`. For other objects, `getFullName()` simply returns the same string as `getName()`. An example:

```
cGate *gate = gate("out", 3);  // out[3]
EV << gate->getName();  // prints "out"
EV << gate->getFullName();  // prints "out[3]"
```

NOTE: When printing out the name of an object, prefer `getFullName()` to `getName()`, especially if the runtime type is not know. This will ensure that the vector index will also be printed if the object has one.

cObject merely defines these member functions, but they return an empty string. Actual storage for a name string and a setName() method is provided by the class cNamedObject, which is also an (indirect) base class for most library classes. Thus, one can assign names to nearly all user-created objects. It it also recommended to do so, because a name makes an object easier to identify in graphical runtimes like Qtenv.

By convention, the object name is the first argument to the constructor of every class, and it defaults to the empty string. To create an object with a name, pass the name string (a const char* pointer) as the first argument of the constructor. For example:

```
cMessage *timeoutMsg = new cMessage("timeout");
```

To change the name of an object, use setName():

```
timeoutMsg->setName("timeout");
```

Both the constructor and setName() make an internal copy of the string, instead of just storing the pointer passed to them.[1]

For convenience and efficiency reasons, the empty string "" and nullptr are treated as interchangeable by library objects. That is, "" is stored as nullptr but returned as "". If one creates a message object with either nullptr or "" as its name string, it will be stored as nullptr, and getName() will return a pointer to a static "".

Hierarchical Name

getFullPath() returns the object's hierarchical name. This name is produced by prepending the *full name* (getFullName()) with the parent or owner object's getFullPath(), separated by a dot. For example, if the out[3] gate in the previous example belongs to a module named classifier, which in turn is part of a network called Queueing, then the gate's getFullPath() method will return "Queueing.classifier.out[3]".

```
cGate *gate = gate("out", 3);  // out[3]
EV << gate->getName();  // prints "out"
EV << gate->getFullName();  // prints "out[3]"
EV << gate->getFullPath();  // prints "Queueing.classifier.out[3]"
```

The getFullName() and getFullPath() methods are extensively used in graphical runtime environments like Qtenv, and also when assembling runtime error messages.

In contrast to getName() and getFullName() which return const char * pointers, getFullPath() returns std::string. This makes no difference when logging via EV«, but when getFullPath() is used as a "%s" argument to sprintf(), one needs to write getFullPath().c_str().

```
char buf[100];
sprintf("msg is '%80s'", msg->getFullPath().c_str()); // note c_str()
```

Class Name

The getClassName() member function returns the class name as a string, including the namespace. getClassName() internally relies on C++ RTTI.

[1] In a simulation, there are usually many objects with the same name: modules, parameters, gates, etc. To conserve memory, several classes keep names in a shared, reference-counted *name pool* instead of making separate copies for each object. The runtime cost of looking up an existing string in the name pool and incrementing its reference count also compares favorably to the cost of allocation and copying.

An example:

```
const char *className = msg->getClassName(); // returns "omnetpp::cMessage"
```

Cloning Objects

The `dup()` member function creates an exact copy of the object, duplicating contained objects also if necessary. This is especially useful in the case of message objects.

```
cMessage *copy = msg->dup();
```

`dup()` delegates to the copy constructor. Classes also declare an assignment operator (`operator=()`) which can be used to copy contents of an object into another object of the same type. `dup()`, the copy constructor and the assignment operator all perform deep coping: objects contained in the copied object will also be duplicated if necessary.

`operator=()` differs from the other two in that it does *not* copy the object's name string, i.e. does not invoke `setName()`. The rationale is that the name string is often used for identifying the particular object instance, as opposed to being considered as part of its contents.

7.1.3 Iterators

There are several container classes in the library (`cQueue`, `cArray` etc.) For many of them, there is a corresponding iterator class that one can use to loop through the objects stored in the container.

For example:

```
cQueue queue;

//...
for (cQueue::Iterator it(queue); !it.end(); ++it) {
    cObject *containedObject = *it;
    //...
}
```

7.1.4 Runtime Errors

When library objects detect an error condition, they throw a C++ exception. This exception is then caught by the simulation environment which pops up an error dialog or displays the error message.

At times it can be useful to be able stop the simulation at the place of the error (just before the exception is thrown) and use a C++ debugger to look at the stack trace and examine variables. Enabling the **debug-on-errors** or the **debugger-attach-on-error** configuration option lets you do that – check it in section 11.12.

7.2 Logging from Modules

In a simulation there are often thousands of modules which simultaneously carry out non-trivial tasks. In order to understand a complex simulation, it is essential to know the inputs

and outputs of algorithms, the information on which decisions are based, and the performed actions along with their parameters. In general, logging facilitates understanding which module is doing what and why.

OMNEST makes logging easy and consistent among simulation models by providing its own C++ API and configuration options. The API provides efficient logging with several predefined log levels, global compile-time and runtime filters, per-component runtime filters, automatic context information, log prefixes and other useful features. In the following sections, we look at how to write log statements using the OMNEST logging API.

7.2.1 Log Output

The exact way log messages are displayed to the user depends on the user interface. In the command-line user interface (Cmdenv), the log is simply written to the standard output. In the Qtenv graphical user interface, the main window has an area for displaying the log output from the currently displayed compound module.

7.2.2 Log Levels

All logging must be categorized into one of the predefined log levels. The assigned log level determines how important and how detailed a log statement is. When deciding which log level is appropriate for a particular log statement, keep in mind that they are meant to be local to components. There's no need for a global agreement among all components, because OMNEST provides per component filtering. Log levels are mainly useful because log output can be filtered based on them.

- `LOGLEVEL_OFF` is not a real log level, it can't be used for actual logging. It is only useful for configuration purposes, it completely disables logging.
- `LOGLEVEL_FATAL` is the highest log level. It should be used for fatal (unrecoverable) errors that prevent the component from further operation. It doesn't mean that the simulation must stop immediately (because in such cases the code should throw a cRuntimeError), but rather that the a component is unable to continue normal operation. For example, a special purpose recording component may be unable to continue recording due to the disk being full.
- `LOGLEVEL_ERROR` should be used for recoverable (non-fatal) errors that allow the component to continue normal operation. For example, a MAC layer protocol component could log unsuccessful packet receptions and unsuccessful packet transmissions using this level.
- `LOGLEVEL_WARN` should be used for exceptional (non-error) situations that may be important for users and rarely occur in the component. For example, a MAC layer protocol component could log detected bit errors using this level.
- `LOGLEVEL_INFO` should be used for high-level protocol specific details that are most likely important for the users of the component. For example, a MAC layer protocol component could log successful packet receptions and successful packet transmissions using this level.
- `LOGLEVEL_DETAIL` should be used for low-level protocol-specific details that may be useful and understandable by the users of the component. These messages may help to

track down various protocol-specific issues without actually looking too deep into the code. For example, a MAC layer protocol component could log state machine updates, acknowledge timeouts and selected back-off periods using this level.

- `LOGLEVEL_DEBUG` should be used for high-level implementation-specific technical details that are most likely important for the developers of the component. These messages may help to debug various issues when one is looking at the code. For example, a MAC layer protocol component could log updates to internal state variables, updates to complex data structures using this level.
- `LOGLEVEL_TRACE` is the lowest log level. It should be used for low-level implementation-specific technical details that are mostly useful for the developers of the component. For example, a MAC layer protocol component could log control flow in loops and if statements, entering/leaving methods and code blocks using this level.

7.2.3 Log Statements

OMNEST provides several C++ macros for the actual logging. Each one of these macros act like a C++ stream, so they can be used similarly to `std::cout` with `operator«` (shift operator).

- `EV_FATAL` for `LOGLEVEL_FATAL`
- `EV_ERROR` for `LOGLEVEL_ERROR`
- `EV_WARN` for `LOGLEVEL_WARN`
- `EV_INFO` for `LOGLEVEL_INFO`
- `EV_DETAIL` for `LOGLEVEL_DETAIL`
- `EV_DEBUG` for `LOGLEVEL_DEBUG`
- `EV_TRACE` for `LOGLEVEL_TRACE`
- `EV` is provided for backward compatibility, and defaults to `EV_INFO`

The actual logging is as simple as writing information into one of these special log streams as follows:

```
EV_ERROR << "Connection to server is lost.\n";
EV_WARN << "Queue is full, discarding packet.\n";
EV_INFO << "Packet received , sequence number = " << seqNum << "." << endl;
EV_TRACE << "routeUnicastPacket(" << packet << ");" << endl;
```

NOTE: It is not recommended to use plain `printf()` or `std::cout` for logging. Output from `EV_INFO` and the other log macros can be controlled more easily from `omnetpp.ini`, and it is more convenient to view using Qtenv.

The above C++ macros work well from any C++ class, including OMNEST modules. In fact, they automatically capture a number of context specific information such as the current event, current simulation time, context module, `this` pointer, source file and line number. The final log lines will be automatically extended with a prefix that is created from the captured information (see section 10.6).

In static class member functions or in non-class member functions an extra EV_STATICCONTEXT macro must be present to make sure that normal log macros compile. [2]

```
void findModule(const char *name, cModule *from)
{
    EV_STATICCONTEXT;
    EV_TRACE << "findModule(" << name << ", " << from << ");" << endl;
```

7.2.4 Log Categories

Sometimes it might be useful to further classify log statements into user defined log categories. In the OMNEST logging API, a log category is an arbitrary string provided by the user.

For example, a module test may check for a specific log message in the test's output. Putting the log statement into the test category ensures that extra care is taken when someone changes the wording in the statement to match the one in the test.

Similarily to the normal C++ log macros, there are separate log macros for each log level which also allow specifying the log category. Their name is the same as the normal variants' but simply extended with the _C suffix. They take the log category as the first parameter before any shift operator calls:

```
EV_INFO_C("test") << "Received " << numPacket << " packets in total.\n";
```

7.2.5 Composition and New lines

Occasionally it's easier to produce a log line using multiple statements. Mostly because some computation has to be done between the parts. This can be achieved by omitting the new line from the log statements which are to be continued. And then subsequent log statements must use the same log level, otherwise an implicit new line would be inserted.

```
EV_INFO << "Line starts here, ";
... // some other code without logging
EV_INFO << "and it continues here" << endl;
```

Assuming a simple log prefix that prints the log level in brakets, the above code fragment produces the following output in Cmdenv:

```
[INFO] Line starts here, and it continues here
```

Sometimes it might be useful to split a line into multiple lines to achieve better formatting. In such cases, there's no need to write multiple log statements. Simply insert new lines into the sequence of shift operator calls:

```
EV_INFO << "First line" << endl << "second line" << endl;
```

In the produced output, each line will have the same log prefix, as shown below:

```
[INFO] First line
[INFO] Second line
```

The OMNEST logging API also supports direct printing to a log stream. This is mainly useful when printing is really complicated algorithmically (e.g. printing a multi-dimensional value). The following code could produce multiple log lines each having the same log prefix.

[2] This is due to that in C++ it is impossible determine at compile-time whether a this pointer is accessible.

```
void Matrix::print(std::stream &output) { ... }
void Matrix::someFunction()
{
   print(EV_INFO);
```

7.2.6 Implementation

OMNEST does its best to optimize the performance of logging. The implementation fully supports conditinal compilation of log statements based on their log level. It automatically checks whether the log is recorded anywhere. It also checks global and per-component runtime log levels. The latter is efficiently cached in the components for subsequent checks. See section 10.6 for more details on how to configure these log levels.

The implementation of the C++ log macros makes use of the fact that the `operator«` is bound more loosely than the conditional operator (`?:`). This solves conditional compilation, and also helps runtime checks by redirecting the output to a `null` stream. Unfortunately the `operator«` calls are still evaluated on the `null` stream, even if the log level is disabled.

Rarely just the computation of log statement parameters may be very expensive, and thus it must be avoided if possible. In this case, it is a good idea to make the log statement conditional on whether the output is actually being displayed or recorded anywhere. The `cEnvir::isLoggingEnabled()` call returns false when the output is disabled, such as in "express" mode. Thus, one can write code like this:

```
if (!getEnvir()->isLoggingEnabled())
    EV_DEBUG << "CRC: " << computeExpensiveCRC(packet) << endl;
```

7.3 Random Number Generators

Random numbers in simulation are usually not really random. Rather, they are produced using deterministic algorithms. Based on some internal state, the algorithm performs some deterministic computation to produce a "random" number and the next state. Such algorithms and their implementations are called *random number generators* or RNGs, or sometimes pseudo random number generators or PRNGs to highlight their deterministic nature. The algorithm's internal state is usually initialized from a smaller *seed* value.

Starting from the same seed, RNGs always produce the same sequence of random numbers. This is a useful property and of great importance, because it makes simulation runs repeatable.

RNGs are rarely used directly, because they produce uniformly distributed random numbers. When non-uniform random numbers are needed, mathematical transformations are used to produce random numbers from RNG input that correspond to specific distributions. This is called random variate generation, and it will be covered in the next section, 7.4.

It is often advantageous for simulations to use random numbers from multiple RNG instances. For example, a wireless network simulation may use one RNG for generating traffic, and another RNG for simulating transmission errors in the noisy wireless channel. Since seeds for individual RNGs can be configured independently, this arrangement allows one e.g. to perform several simulation runs with the same traffic but with bit errors occurring in different places. A simulation technique called *variance reduction* is also related to the use of different

random number streams. OMNEST makes it easy to use multiple RNGs in various flexible configurations.

When assigning seeds, it is important that different RNGs and also different simulation runs use non-overlapping series of random numbers. Overlap in the generated random number sequences can introduce unwanted correlation in the simulation results.

7.3.1 RNG Implementations

OMNEST comes with the following RNG implementations.

Mersenne Twister

By default, OMNEST uses the Mersenne Twister RNG (MT) by M. Matsumoto and T. Nishimura [MN98]. MT has a period of $2^{19937} - 1$, and 623-dimensional equidistribution property is assured. MT is also very fast: as fast or faster than ANSI C's `rand()`.

The "Minimal Standard" RNG

OMNEST releases prior to 3.0 used a linear congruential generator (LCG) with a cycle length of $2^{31} - 2$, described in [Jai91], pp. 441-444,455. This RNG is still available and can be selected from `omnetpp.ini` (Chapter 11). This RNG is only suitable for small-scale simulation studies. As shown by Karl Entacher et al. in [EHW02], the cycle length of about 2^{31} is too small (on todays fast computers it is easy to exhaust all random numbers), and the structure of the generated "random" points is too regular. The [Hel98] paper provides a broader overview of issues associated with RNGs used for simulation, and it is well worth reading. It also contains useful links and references on the topic.

The Akaroa RNG

When a simulation is executed under Akaroa control (see section 11.20), it is also possible to let OMNEST use Akaroa's RNG. This needs to be configured in `omnetpp.ini` (section 10.5).

Other RNGs

OMNEST allows plugging in your own RNGs as well. This mechanism, based on the `cRNG` interface, is described in section 17.5. For example, one candidate to include could be L'Ecuyer's CMRG [LSCK02] which has a period of about 2^{191} and can provide a large number of *guaranteed* independent streams.

7.3.2 Global and Component-Local RNGs

OMNEST can be configured to make several RNGs available for the simulation model. These *global* or *physical* RNGs are numbered from 0 to $numRNGs - 1$, and can be seeded independently.

However, usually model code doesn't directly work with those RNGs. Instead, there is an indirection step introduced for additional flexibility. When random numbers are drawn in a model, the code usually refers to *component-local* or *logical* RNG numbers. These local RNG

numbers are mapped to global RNG indices to arrive at actual RNG instances. This mapping occurs on per-component basis. That is, each module and channel object contains a mapping table similar to the following:

Local RNG index		Global RNG
0	→	0
1	→	0
2	→	2
3	→	1
4	→	1
5	→	3

In the example, the module or channel in question has 6 local (logical) RNGs that map to 4 global (physical) RNGs.

NOTE: Local RNG number 0 is special in the sense that all random number functions use that RNG, unless explicitly told otherwise by specifying an *rng=k* argument.

The local-to-global mapping, as well as the number of number of global RNGs and their seeding can be configured in `omnetpp.ini` (see section 10.5).

The mapping can be set up arbitrarily, with the default being identity mapping (that is, local RNG k refers to global RNG k.) The mapping allows for flexibility in RNG and random number streams configuration – even for simulation models which were not written with RNG awareness. For example, even if modules in a simulation only use the default, local RNG number 0, one can set up mapping so that different groups of modules use different physical RNGs.

In theory, RNGs could also be instantiated and used directly from C++ model code. However, doing so is not recommended, because the model would lose configurability via `omnetpp.ini`.

7.3.3 Accessing the RNGs

RNGs are represented with subclasses of the abstract class `cRNG`. In addition to random number generation methods like `intRand()` and `doubleRand()`, the `cRNG` interface also includes methods like `selfTest()` for basic integrity checking and `getNumbersDrawn()` to query the number of random numbers generated.

RNGs can be accessed by local RNG number via `cComponent`'s `getRNG(k)` method. To access global global RNGs directly by their indices, one can use `cEnvir`'s `getRNG(k)` method. However, RNGs rarely need to be accessed directly. Most simulations will only use them via random variate generation functions, described in the next section.

7.4 Generating Random Variates

Random numbers produced by RNGs are uniformly distributed. This section describes how to obtain streams of non-uniformly distributed random numbers from various distributions.

The simulation library supports the following distributions:

Distribution	**Description**
Continuous distributions	

***uniform**(a, b)*	uniform distribution in the range [a,b)
***exponential**(mean)*	exponential distribution with the given mean
***normal**(mean, stddev)*	normal distribution with the given mean and standard deviation
***truncnormal**(mean, stddev)*	normal distribution truncated to nonnegative values
***gamma_d**(alpha, beta)*	gamma distribution with parameters alpha>0, beta>0
***beta**(alpha1, alpha2)*	beta distribution with parameters alpha1>0, alpha2>0
***erlang_k**(k, mean)*	Erlang distribution with k>0 phases and the given mean
***chi_square**(k)*	chi-square distribution with k>0 degrees of freedom
***student_t**(i)*	student-t distribution with i>0 degrees of freedom
***cauchy**(a, b)*	Cauchy distribution with parameters a,b where b>0
***triang**(a, b, c)*	triangular distribution with parameters a<=b<=c, a!=c
***lognormal**(m, s)*	lognormal distribution with mean m and variance s>0
***weibull**(a, b)*	Weibull distribution with parameters a>0, b>0
***pareto_shifted**(a, b, c)*	generalized Pareto distribution with parameters a, b and shift c
Discrete distributions	
***intuniform**(a, b)*	uniform integer from a..b
***bernoulli**(p)*	result of a Bernoulli trial with probability 0<=p<=1 (1 with probability p and 0 with probability (1-p))
***binomial**(n, p)*	binomial distribution with parameters n>=0 and 0<=p<=1
***geometric**(p)*	geometric distribution with parameter 0<=p<=1
***negbinomial**(n, p)*	negative binomial distribution with parameters n>0 and 0<=p<=1
***poisson**(lambda)*	Poisson distribution with parameter lambda

Some notes:

- *intuniform()* generates integers including both the lower and upper limit, so for example the outcome of tossing a coin could be written as `intuniform(1,2)`.
- *truncnormal()* is the normal distribution truncated to nonnegative values; its implementation generates a number with normal distribution and if the result is negative, it keeps generating other numbers until the outcome is nonnegative.

There are several ways to generate random numbers from these distributions, as described in the next sections.

7.4.1 Component Methods

The preferred way is to use methods defined on `cComponent`, the common base class of modules and channels:

```
double uniform(double a, double b, int rng=0) const;
```

```
double exponential(double mean, int rng=0) const;
double normal(double mean, double stddev, int rng=0) const;
...
```

These methods work with the component's local RNGs, and accept the RNG index (default 0) in their extra `int` parameter.

Since most simulation code is located in methods of simple modules, these methods can be usually called in a concise way, without an explicit module or channel pointer. An example:

```
scheduleAt(simTime() + exponential(1.0), msg);
```

There are two additional methods, `intrand()` and `dblrand()`. `intrand(n)` generates random integers in the range $[0, n-1]$, and `dblrand()` generates a random double on $[0, 1)$. They also accept an additional local RNG index that defaults to 0.

7.4.2 Random Number Stream Classes

It is sometimes useful to be able to pass around random variate generators as objects. The classes `cUniform`, `cExponential`, `cNormal`, etc. fulfill this need.

These classes subclass from the `cRandom` abstract class. `cRandom` was designed to encapsulate random number streams. Its most important method is `draw()` that returns a new random number from the stream. `cUniform`, `cExponential` and other classes essentially bind the distribution's parameters and an RNG to the generation function.

Figure 7.2: Random number stream classes

Let us see for example `cNormal`. The constructor expects an RNG (`cRNG` pointer) and the parameters of the distribution, mean and standard deviation. It also has a default constructor, as it is a requirement for `Register_Class()`. When the default constructor is used, the parameters can be set with `setRNG()`, `setMean()` and `setStddev()`. `setRNG()` is defined on `cRandom`. The `draw()` method, of course, is redefined to return a random number from the normal distribution.

An example that shows the use of a random number stream as an object:

```
cNormal *normal = new cNormal(getRNG(0), 0, 1); // unit normal distr.
printRandomNumbers(normal, 10);
...

void printRandomNumbers(cRandom *rand, int n)
{
    EV << "Some numbers from a " << rand->getClassName() << ":" << endl;
    for (int i = 0; i < n; i++)
        EV << rand->draw() << endl;
}
```

Another important property of cRandom is that it can encapsulate state. That is, subclasses can be implemented that, for example, return autocorrelated numbers, numbers from a stochastic process, or simply elements of a stored sequence (e.g. one loaded from a trace file).

7.4.3 Generator Functions

Both the cComponent methods and the random number stream classes described above have been implemented with the help of standalone generator functions. These functions take a cRNG pointer as their first argument.

```
double uniform(cRNG *rng, double a, double b);
double exponential(cRNG *rng, double mean);
double normal(cRNG *rng, double mean, double stddev);
...
```

7.4.4 Random Numbers from Histograms

One can also specify a distribution as a histogram. The cHistogram, cKSplit and cPSquare classes can be used to generate random numbers from histograms. This feature is documented later, with the statistical classes.

7.4.5 Adding New Distributions

One can easily add support for new distributions. We recommend that you write a standalone generator function first. Then you can add a cRandom subclass that wraps it, and/or module (channel) methods that invoke it with the module's local RNG. If the function is registered with the Define_NED_Function() macro (see 7.12), it will be possible to use the new distribution in NED files and ini files, as well.

If you need a random number stream that has state, you need to subclass from cRandom.

7.5 Container Classes

7.5.1 Queue class: cQueue

Basic Usage

cQueue is a container class that acts as a queue. cQueue can hold objects of type derived from cObject (almost all classes from the OMNEST library), such as cMessage, cPar, etc. Normally, new elements are inserted at the back, and removed from the front.

The member functions dealing with insertion and removal are insert() and pop().

```
cQueue queue("my-queue");
cMessage *msg;

// insert messages
for (int i = 0; i < 10; i++) {
```

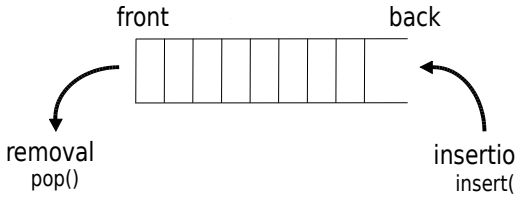


Figure 7.3: cQueue: insertion and removal

```
    msg = new cMessage;
    queue.insert(msg);
}

// remove messages
while(!queue.isEmpty()) {
    msg = (cMessage *)queue.pop();
    delete msg;
}
```

The `length()` member function returns the number of items in the queue, and `empty()` tells whether there is anything in the queue.

There are other functions dealing with insertion and removal. The `insertBefore()` and `insertAfter()` functions insert a new item exactly before or after a specified one, regardless of the ordering function.

The `front()` and `back()` functions return pointers to the objects at the front and back of the queue, without affecting queue contents.

The `pop()` function can be used to remove items from the tail of the queue, and the `remove()` function can be used to remove any item known by its pointer from the queue:

```
queue.remove(msg);
```

Priority Queue

By default, `cQueue` implements a FIFO, but it can also act as a priority queue, that is, it can keep the inserted objects ordered. To use this feature, one needs to provide a comparison function that takes two `cObject` pointers, and returns -1, 0 or 1 (see the reference for details). An example of setting up an ordered `cQueue`:

```
cQueue queue("queue", someCompareFunc);
```

If the queue object is set up as an ordered queue, the `insert()` function uses the ordering function: it searches the queue contents from the head until it reaches the position where the new item needs to be inserted, and inserts it there.

Iterators

The `cQueue::Iterator` class lets one iterate over the contents of the queue and examine each object.

The `cQueue::Iterator` constructor expects the queue object in the first argument. Normally, forward iteration is assumed, and the iteration is initialized to point at the front of the queue. For reverse iteration, specify `reverse=true` as the optional second argument. After that, the class acts as any other OMNEST iterator: one can use the `++` and `--` operators to advance it, the `*` operator to get a pointer to the current item, and the `end()` member function to examine whether the iterator has reached the end (or the beginning) of the queue.

Forward iteration:

```
for (cQueue::Iterator iter(queue); !iter.end(), iter++) {
    cMessage *msg = (cMessage *) *iter;
    //...
}
```

Reverse iteration:

```
for (cQueue::Iterator iter(queue, true); !iter.end(), iter--) {
    cMessage *msg = (cMessage *) *iter;
    //...
}
```

7.5.2 Expandable Array: cArray

Basic Usage

`cArray` is a container class that holds objects derived from `cObject`. `cArray` implements a dynamic-size array: its capacity grows automatically when it becomes full. `cArray` stores pointers of objects inserted instead of making copies.

Creating an array:

```
cArray array("array");
```

Adding an object at the first free index:

```
cMsgPar *p = new cMsgPar("par");
int index = array.add(p);
```

Adding an object at a given index (if the index is occupied, you will get an error message):

```
cMsgPar *p = new cMsgPar("par");
int index = array.addAt(5,p);
```

Finding an object in the array:

```
int index = array.find(p);
```

Getting a pointer to an object at a given index:

```
cPar *p = (cPar *) array[index];
```

You can also search the array or get a pointer to an object by the object's name:

```
int index = array.find("par");
Par *p = (cPar *) array["par"];
```

You can remove an object from the array by calling `remove()` with the object name, the index position or the object pointer:

```
array.remove("par");
array.remove(index);
array.remove(p);
```

The `remove()` function doesn't deallocate the object, but it returns the object pointer. If you also want to deallocate it, you can write:

```
delete array.remove(index);
```

Iteration

`cArray` has no iterator, but it is easy to loop through all the indices with an integer variable. The `size()` member function returns the largest index plus one.

```
for (int i = 0; i < array.size(); i++) {
  if (array[i]) { // is this position used?
    cObject *obj = array[i];
    EV << obj->getName() << endl;
  }
}
```

7.6 Routing Support: cTopology

7.6.1 Overview

The `cTopology` class was designed primarily to support routing in communication networks.

A `cTopology` object stores an abstract representation of the network in a graph form:

- each `cTopology` node corresponds to a *module* (simple or compound), and
- each `cTopology` edge corresponds to a *link* or *series of connecting links*.

One can specify which modules to include in the graph. Compound modules may also be selected. The graph will include all connections among the selected modules. In the graph, all nodes are at the same level; there is no submodule nesting. Connections which span across compound module boundaries are also represented as one graph edge. Graph edges are directed, just as module gates are.

If you are writing a router or switch model, the `cTopology` graph can help you determine what nodes are available through which gate and also to find optimal routes. The `cTopology` object can calculate shortest paths between nodes for you.

The mapping between the graph (nodes, edges) and network model (modules, gates, connections) is preserved: one can find the corresponding module for a `cTopology` node and vice versa.

7.6.2 Basic Usage

One can extract the network topology into a `cTopology` object with a single method call. There are several ways to specify which modules should be included in the topology:

- by module type
- by a parameter's presence and value
- with a user-supplied boolean function

First, you can specify which node types you want to include. The following code extracts all modules of type `Router` or `Host`. (`Router` and `Host` can be either simple or compound module types.)

```
cTopology topo;
topo.extractByModuleType("Router", "Host", nullptr);
```

Any number of module types can be supplied; the list must be terminated by `nullptr`.

A dynamically assembled list of module types can be passed as a `nullptr`-terminated array of `const char*` pointers, or in an STL string vector `std::vector<std::string>`. An example for the former:

```
cTopology topo;
const char *typeNames[3];
typeNames[0] = "Router";
typeNames[1] = "Host";
typeNames[2] = nullptr;
topo.extractByModuleType(typeNames);
```

Second, you can extract all modules which have a certain parameter:

```
topo.extractByParameter("ipAddress");
```

You can also specify that the parameter must have a certain value for the module to be included in the graph:

```
cMsgPar yes = "yes";
topo.extractByParameter("includeInTopo", &yes);
```

The third form allows you to pass a function which can determine for each module whether it should or should not be included. You can have `cTopology` pass supplemental data to the function through a `void*` pointer. An example which selects all top-level modules (and does not use the `void*` pointer):

```
int selectFunction(cModule *mod, void *)
{
  return mod->getParentModule() == getSimulation()->getSystemModule();
}

topo.extractFromNetwork(selectFunction, nullptr);
```

A `cTopology` object uses two types: `cTopology::Node` for nodes and `cTopology::Link` for edges. (`cTopology::LinkIn` and `cTopology::LinkOut` are aliases for `cTopology::Link`; we'll talk about them later.)

Once you have the topology extracted, you can start exploring it. Consider the following code (we'll explain it shortly):

```
for (int i = 0; i < topo.getNumNodes(); i++) {
  cTopology::Node *node = topo.getNode(i);
```

```
    EV << "Node i=" << i << " is " << node->getModule()->getFullPath() << endl;
    EV << " It has " << node->getNumOutLinks() << " conns to other nodes\n";
    EV << " and " << node->getNumInLinks() << " conns from other nodes\n";

    EV << " Connections to other modules are:\n";
    for (int j = 0; j < node->getNumOutLinks(); j++) {
      cTopology::Node *neighbour = node->getLinkOut(j)->getRemoteNode();
      cGate *gate = node->getLinkOut(j)->getLocalGate();
      EV << " " << neighbour->getModule()->getFullPath()
         << " through gate " << gate->getFullName() << endl;
    }
  }
```

The `getNumNodes()` member function returns the number of nodes in the graph, and `getNode(i)` returns a pointer to the *i*th node, a `cTopology::Node` structure.

The correspondence between a graph node and a module can be obtained by `getNodeFor()` method:

```
cTopology::Node *node = topo.getNodeFor(module);
cModule *module = node->getModule();
```

The `getNodeFor()` member function returns a pointer to the graph node for a given module. (If the module is not in the graph, it returns `nullptr`). `getNodeFor()` uses binary search within the `cTopology` object so it is relatively fast.

`cTopology::Node`'s other member functions let you determine the connections of this node: `getNumInLinks()`, `getNumOutLinks()` return the number of connections, `getLinkIn(i)` and `getLinkOut(i)` return pointers to graph edge objects.

By calling member functions of the graph edge object, you can determine the modules and gates involved. The `getRemoteNode()` function returns the other end of the connection, and `getLocalGate()`, `getRemoteGate()`, `getLocalGateId()` and `getRemoteGateId()` return the gate pointers and ids of the gates involved. (Actually, the implementation is a bit tricky here: the same graph edge object `cTopology::Link` is returned either as `cTopology::LinkIn` or as `cTopology::LinkOut` so that "remote" and "local" can be correctly interpreted for edges of both directions.)

7.6.3 Shortest Paths

The real power of `cTopology` is in finding shortest paths in the network to support optimal routing. `cTopology` finds shortest paths from *all* nodes *to* a target node. The algorithm is computationally inexpensive. In the simplest case, all edges are assumed to have the same weight.

A real-life example assumes we have the target module pointer; finding the shortest path to the target looks like this:

```
cModule *targetmodulep =...;
cTopology::Node *targetnode = topo.getNodeFor(targetmodulep);
topo.calculateUnweightedSingleShortestPathsTo(targetnode);
```

This performs the Dijkstra algorithm and stores the result in the `cTopology` object. The result can then be extracted using `cTopology` and `cTopology::Node` methods. Naturally, each call

to `calculateUnweightedSingleShortestPathsTo()` overwrites the results of the previous call.

Walking along the path from our module to the target node:

```
cTopology::Node *node = topo.getNodeFor(this);

if (node == nullptr) {
  EV << "We (" << getFullPath() << ") are not included in the topology.\n";
}
else if (node->getNumPaths()==0) {
  EV << "No path to destination.\n";
}
else {
  while (node != topo.getTargetNode()) {
    EV << "We are in " << node->getModule()->getFullPath() << endl;
    EV << node->getDistanceToTarget() << " hops to go\n";
    EV << "There are " << node->getNumPaths()
       << " equally good directions, taking the first one\n";
    cTopology::LinkOut *path = node->getPath(0);
    EV << "Taking gate " << path->getLocalGate()->getFullName()
       << " we arrive in " << path->getRemoteNode()->getModule()->getFullPath()
       << " on its gate " << path->getRemoteGate()->getFullName() << endl;
    node = path->getRemoteNode();
  }
}
```

The purpose of the `getDistanceToTarget()` member function of a node is self-explanatory. In the unweighted case, it returns the number of hops. The `getNumPaths()` member function returns the number of edges which are part of a shortest path, and `path(i)` returns the *i*th edge of them as `cTopology::LinkOut`. If the shortest paths were created by the `...SingleShortestPaths()` function, `getNumPaths()` will always return 1 (or 0 if the target is not reachable), that is, only one of the several possible shortest paths are found. The `...MultiShortestPathsTo()` functions find all paths, at increased run-time cost. The `cTopology`'s `getTargetNode()` function returns the target node of the last shortest path search.

You can enable/disable nodes or edges in the graph. This is done by calling their `enable()` or `disable()` member functions. Disabled nodes or edges are ignored by the shortest paths calculation algorithm. The `isEnabled()` member function returns the state of a node or edge in the topology graph.

One usage of `disable()` is when you want to determine in how many hops the target node can be reached from our node *through a particular output gate*. To compute this, you compute the shortest paths to the target *from the neighbor node* while disabling the current node to prevent the shortest paths from going through it:

```
cTopology::Node *thisnode = topo.getNodeFor(this);
thisnode->disable();
topo.calculateUnweightedSingleShortestPathsTo(targetnode);
thisnode->enable();

for (int j = 0; j < thisnode->getNumOutLinks(); j++) {
  cTopology::LinkOut *link = thisnode->getLinkOut(i);
  EV << "Through gate " << link->getLocalGate()->getFullName() << " : "
     << 1 + link->getRemoteNode()->getDistanceToTarget() << " hops" << endl;
```

```
}
```

In the future, other shortest path algorithms will also be implemented:

```
unweightedMultiShortestPathsTo(cTopology::Node *target);
weightedSingleShortestPathsTo(cTopology::Node *target);
weightedMultiShortestPathsTo(cTopology::Node *target);
```

7.6.4 Manipulating the graph

`cTopology` also has methods that let one manipulate the stored graph, or even, build a graph from scratch. These methods are `addNode()`, `deleteNode()`, `addLink()` and `deleteLink()`.

When extracting the topology from the network, `cTopology` uses the factory methods `createNode()` and `createLink()` to instantiate the node and link objects. These methods may be overridden by subclassing `cTopology` if the need arises, for example when it is useful to be able to store additional information in those objects.

7.7 Pattern Matching

Since version 4.3, OMNEST contains two utility classes for pattern matching, `cPatternMatcher` and `cMatchExpression`.

`cPatternMatcher` is a glob-style pattern matching class, adopted to special OMNEST requirements. It recognizes wildcards, character ranges and numeric ranges, and supports options such as case sensitive and whole string matching. `cMatchExpression` builds on top of `cPatternMatcher` and extends it in two ways: first, it lets you combine patterns with AND, OR, NOT into boolean expressions, and second, it applies the pattern expressions to *objects* instead of text. These classes are especially useful for making model-specific configuration files more concise or more powerful by introducing patterns.

7.7.1 cPatternMatcher

`cPatternMatcher` holds a pattern string and several option flags, and has a `matches()` boolean function that determines whether the string passed as argument matches the pattern with the given flags. The pattern and the flags can be set via the constructor or by calling the `setPattern()` member function.

The pattern syntax is a variation on Unix *glob*-style patterns. The most apparent differences to globbing rules are the distinction between `*` and `**`, and that character ranges should be written with curly braces instead of square brackets; that is, *any-letter* is expressed as `{a-zA-Z}` and not as `[a-zA-Z]`, because square brackets are reserved for the notation of module vector indices.

The following option flags are supported:

- *dottedpath*: controls whether some wildcards (`?`, `*`) will match dots
- *fullstring*: controls whether to do full string or substring match.
- *casesensitive*: whether matching is case sensitive or case insensitive

Patterns may contain the following elements:

- *question mark*, `?` : matches any character (except dot if *dottedpath*=true)
- *asterisk*, `*` : matches zero or more characters (except dots if *dottedpath*=true)
- *double asterisk*, `**` : matches zero or more characters, including dots
- *set*, e.g. `{a-zA-Z}` : matches any character that is contained in the set
- *negated set*, e.g. `{^a-z}`: matches any character that is NOT contained in the set
- *numeric range*, e.g. `{38..150}` : matches any number (i.e. sequence of digits) in the given range
- *numeric index range*, e.g. `[38..150]` : matches any number in square brackets in the given range
- *backslash*, `\` : takes away the special meaning of the subsequent character

NOTE: The *dottedpath* option was introduced to make matching OMNEST module paths more powerful. When it is off (*dottedpath*=false), there is no difference between `*` and `**`, they both match any character sequence. However, when matching OMNEST module paths or other strings where dot is a separator character, it is useful to turn on the dottedpath mode (*dottedpath*=true). In that mode, `*`, not being able to cross a dot, can match only a single path component (or part of it), and `**` can match multiple path components.

Sets and negated sets can contain several character ranges and also enumeration of characters, for example `{_a-zA-Z0-9}` or `{xyzc-f}`. To include a hyphen in the set, place it at a position where it cannot be interpreted as character range, for example `{a-z-}` or `{-a-z}`. To include a close brace in the set, it must be the first character: `{}a-z}`, or for a negated set: `{^}a-z}`. A backslash is always taken as literal backslash (and NOT as escape character) within set definitions. When doing case-insensitive match, avoid ranges that include both alpha and non-alpha characters, because they might cause funny results.

For numeric ranges and numeric index ranges, ranges are inclusive, and both the start and the end of the range are optional; that is, `{10..}`, `{..99}` and `{..}` are all valid numeric ranges (the last one matches any number). Only nonnegative integers can be matched. Caveat: `{17..19}` will match `"a17"`, `"117"` and also `"963217"`!

The `cPatternMatcher` constructor and the `setPattern()` member function have similar signatures:

```
cPatternMatcher(const char *pattern, bool dottedpath, bool fullstring,
                bool casesensitive);
void setPattern(const char *pattern, bool dottedpath, bool fullstring,
                bool casesensitive);
```

The matcher function:

```
bool matches(const char *text);
```

There are also some more utility functions for printing the pattern, determining whether a pattern contains wildcards, etc.

Example:

```
cPatternMatcher matcher("**.host[*]", true, true, true);
EV << matcher.matches("Net.host[0]") << endl;  // -> true
EV << matcher.matches("Net.area1.host[0]") << endl;  // -> true
EV << matcher.matches("Net.host") << endl;  // -> false
EV << matcher.matches("Net.host[0].tcp") << endl;  // -> false
```

7.7.2 cMatchExpression

The `cMatchExpression` class builds on top of `cPatternMatcher`, and lets one determine whether an *object* matches a given pattern expression.

A pattern expression consists of elements in the *fieldname =~ pattern* syntax; they check whether the string representation of the given field of the object matches the pattern. [3] For example, `srcAddr(192.168.0.*)` will match if the *srcAddr* field of the object starts with `192.168.0.`. A naked *pattern* (without the field name and the `=~` operator) is also accepted, and it will be matched against the *default field* of the object, which will usually be its name.

These elements can be combined with the AND, OR, NOT operators, accepted in both lowercase and uppercase. AND has higher precedence than OR, but parentheses can be used to change the evaluation order.

Pattern examples:

- `"node*"`
- `"node* or host*"`
- `"packet-* and className =~ PPPFrame"`
- `"className =~ TCPSegment and byteLength =~ {4096..}"`
- `"className=~TCPSegment and (SYN or DATA-*) and not kind=~{0..2}"`

The `cMatchExpression` class has a constructor and `setPattern()` method similar to those of `cPatternMatcher`:

```
cMatchExpression(const char *pattern, bool dottedpath, bool fullstring,
                 bool casesensitive);
void setPattern(const char *pattern, bool dottedpath, bool fullstring,
                bool casesensitive);
```

However, the matcher function takes a `cMatchExpression::Matchable` instead of string:

```
bool matches(const Matchable *object);
```

This means that objects to be matched must either be subclassed from `cMatchExpression::Matchable`, or be wrapped into some adapter class that does. `cMatchExpression::Matchable` is a small abstract class with only a few pure virtual functions:

```
/**
 * Objects to be matched must implement this interface
 */
class SIM_API Matchable
```

[3] Note that the syntax has changed in OMNEST version 6.0. In prior versions, field matchers had to be written as *fieldname(pattern)*.

```
{
  public:
    /**
     * Return the default string to match. The returned pointer will not be
     * cached by the caller, so it is OK to return a pointer to a static buffer.
     */
    virtual const char *getAsString() const = 0;

    /**
     * Return the string value of the given attribute, or nullptr if the object
     * doesn't have an attribute with that name. The returned pointer will not
     * be cached by the caller, so it is OK to return a pointer to a static buffer
     */
    virtual const char *getAsString(const char *attribute) const = 0;

    /**
     * Virtual destructor.
     */
    virtual ~Matchable() {}
};
```

To be able to match instances of an existing class that is not already a `Matchable`, one needs to write an adapter class. An adapter class that we can look at as an example is `cMatchableString`. `cMatchableString` makes it possible to match strings with a `cMatchExpression`, and is part of OMNEST:

```
/**
 * Wrapper to make a string matchable with cMatchExpression.
 */
class cMatchableString : public cMatchExpression::Matchable
{
  private:
    std::string str;
  public:
    cMatchableString(const char *s) {str = s;}
    virtual const char *getAsString() const {return str.c_str();}
    virtual const char *getAsString(const char *name) const {return nullptr;}
};
```

An example:

```
cMatchExpression expr("foo* or bar*", true, true, true);
cMatchableString str1("this is a foo");
cMatchableString str2("something else");
EV << expr.matches(&str1) << endl; // -> true
EV << expr.matches(&str2) << endl; // -> false
```

Or, by using temporaries:

```
EV << expr.matches(&cMatchableString("this is a foo")) << endl; // -> true
EV << expr.matches(&cMatchableString("something else")) << endl; // -> false
```

7.8 Dynamic Expression Evaluation

The NED **expr()** operator encapsulates a formula in an object form. On the C++ side, the object is an instance of cOwnedDynamicExpression.

The expression can be evaluated using the evalute() method that returns a cValue, or one of typed methods: boolValue(), intValue(), doubleValue(), stringValue(), xmlValue(). But before that, a custom resolver needs to be implemented, and installed using the setResolver(). The resolver subclasses from cDynamicExpression::IResolver, and its methods readVariable(), readMember(), callFunction(), callMethod() determine how to evaluate various constructs in the expression.

7.9 Collecting Summary Statistics and Histograms

There are several statistic and result collection classes: cStdDev, cHistogram, cPSquare and cKSplit. They are all derived from the abstract base class cStatistic; histogram-like classes derive from cAbstractHistogram.[4]

- cStdDev keeps summary statistics (mean, standard deviation, range) of weighted or unweighted observations.
- cHistogram is for collecting observations into a histogram. cHistogram is highly configurable, supports adding/removing/merging bins dynamically, and can produce a good histogram from most distributions without requiring manual configuration.
- cPSquare is a class that uses the P^2 algorithm described in [JC85]. The algorithm calculates quantiles without storing the observations; one can also think of it as a histogram with equiprobable cells.
- cKSplit is adaptive histogram-like algorithm which performs dynamic subdivision of the bins to refine resolution at the bulk of the distribution.

All classes use the double type for representing observations, and compute all metrics in the same data type (except the observation count, which is int64_t.)

For weighted statistics, weights are also doubles. Being able to handle non-integer weights is important because weighted statistics are often used for computing time averages, e.g. average queue length or average channel utilization.

7.9.1 cStdDev

The cStdDev class is meant to collect summary statistics of observations. If you also need to compute a histogram, use cHistogram (or cKSplit/cPSquare) instead, because those classes already include the functionality of cStdDev.

cStdDev can collect unweighted or weighted statistics. This needs to be decided in the constructor call, and cannot be changed later. Specify true as the second argument for weighted statistics.

[4] Earlier versions of OMNEST had more statistical classes: cWeightedStdDev, cLongHistogram, cDoubleHistogram, cVarHistogram. The functionality of these classes have been consolidated into cStdDev and cHistogram.

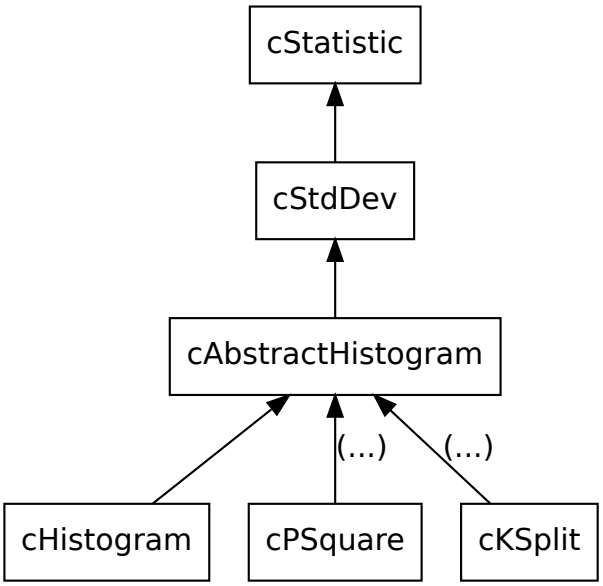


Figure 7.4: Statistics classes

```
cStdDev unweighted("packetDelay");  // unweighted
cStdDev weighted("queueLength", true); // weighted
```

Observations are added to the statistics by using the `collect()` or the `collectweighted()` methods. The latter takes two parameters, the value and the weight.

```
for (double value : values)
    unweighted.collect(value);

for (double value : values2) {
    double weight = ...
    weighted.collectWeighted(value, weight);
}
```

Statistics can be obtained from the object with the following methods: `getCount()`, `getMin()`, `getMax()`, `getMean()`, `getStddev()`, `getVariance()`.

There are two getter methods that only work for unweighted statistics: `getSum()` and `getSqrSum()`. Plain (unweighted) sum and sum of squares are not computed for weighted observations, and it is an error to call these methods in the weighted case.

Other getter methods are primarily meant for weighted statistics: `getSumWeights()`, `getWeightedSum()`, `getSqrSumWeights()`, `getWeightedSqrSum()`. When called on unweighted statistics, these methods simply assume a weight of 1.0 for all observations.

An example:

```
EV << "count = " << unweighted.getCount() << endl;
EV << "mean = " << unweighted.getMean() << end;
EV << "stddev = " << unweighted.getStddev() << end;
EV << "min = " << unweighted.getMin() << end;
EV << "max = " << unweighted.getMax() << end;
```

7.9.2 cHistogram

cHistogram is able to represent both uniform and non-uniform bin histograms, and supports both weighted and unweighted observations. The histogram can be modified dynamically: it can be extended with new bins, and adjacent bins can be merged. In addition to the bin values (which mean count in the unweighted case, and sum of weights in the weighted case), the histogram object also keeps the number (or sum of weights) of the lower and upper outliers ("underflows" and "overflows".)

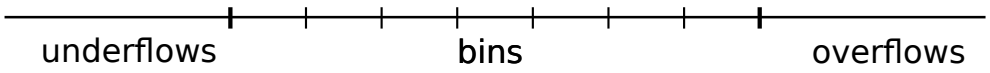


Figure 7.5: Histograms keep track of outliers as well

Setting up and managing the bins based on the collected observations is usually delegated to a strategy object. However, for most use cases, histogram strategies is not something the user needs to be concerned with. The default constructor of cHistogram sets up the histogram with a default strategy that usually produces a good quality histogram without requiring manual configuration or a-priori knowledge about the distribution. For special use cases, there are other histogram strategies, and it is also possible to write new ones.

Creating a Histogram

cHistogram has several constructors variants. Like with cStdDev, it needs to be decided in the constructor call by a boolean argument whether the histogram should collect unweighted (false) or weighted (true) statistics; the default is unweighted. Another argument is a number of bins hint. (The actual number of bins produced might slightly differ, due to dynamic range extensions and bin merging performed by some strategies.)

```
cHistogram unweighted1("packetDelay");  // unweighted
cHistogram unweighted2("packetDelay", 10);  // unweighted, with ~10 bins
cHistogram weighted1("queueLength", true); // weighted
cHistogram weighted2("queueLength", 10, true); // weighted, with ~10 bins
```

It is also possible to provide a strategy object in a constructor call. (The strategy object may also be set later though, using setStrategy(). It must be called before the first observation is collected.)

```
cHistogram autoRangeHist("queueLength", new cAutoRangeHistogramStrategy());
```

This constructor can also be used to create a histogram without a strategy object, which is useful if you want to set up the histogram bins manually.

```
cHistogram hist("queueLength", nullptr, true); // weighted, no strategy
```

cHistogram also has methods where you can provide constraints and hints for setting up the bins: setMode(), setRange(), setRangeExtensionFactor(), setAutoExtend(), setNumBinsHint(), setBinSizeHint(). These methods delegate to similar methods of cAutoRangeHistogramStrategy.

Collecting Observations

Observations are added to the histogram in the same way as with `cStdDev`: using the `collect()` and `collectWeighted()` methods.

Querying the Bins

Histogram bins can be accessed with three member functions: `getNumBins()` returns the number of bins, `getBinEdge(int k)` returns the *k*th bin edge, `getBinValue(int k)` returns the count or sum of weights in bin *k*, and `getBinPDF(int k)` returns the PDF value in the bin (i.e. between `getBinEdge(k)` and `getBinEdge(k+1)`). The `getBinInfo(k)` method returns multiple bin data (edges, value, relative frequency) packed together in a struct. Four other methods, `getUnderflowSumWeights()`, `getOverflowSumWeights()`, `getNumUnderflows()`, `getNumOverflows()`, provide access to the outliers.

These functions, being defined on `cHistogramBase`, are not only available on `cHistogram` but also for `cPSquare` and `cKSplit`.

For `cHistogram`, bin edges and bin values can also be accessed as a vector of `double`s, using the `getBinEdges()` and `getBinValues()` methods.

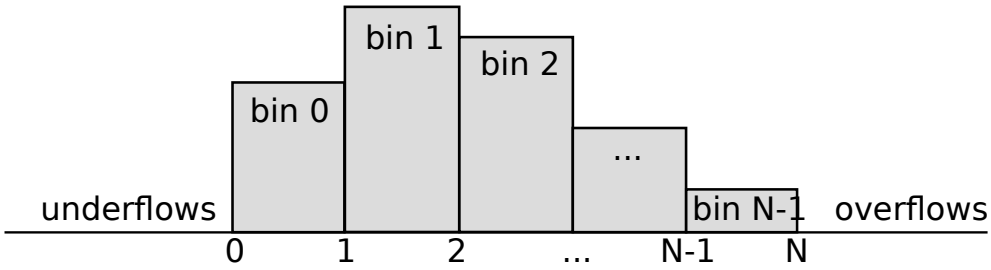


Figure 7.6: Bin edges and bins of an N-bin histogram

An example:

```
EV << "[" << hist.getMin() << "," << hist.getBinEdge(0) << "): "
   << hist.getUnderflowSumWeights() << endl;
int numBins = hist.getNumBins();
for (int i = 0; i < numBins; i++) {
  EV << "[" << hist.getBinEdge(i) << "," << hist.getBinEdge(i+1) << "): "
     << hist.getBinValue(i) << endl;
}
EV << "[" << hist.getBinEdge(numBins) << "," << hist.getMax() << "]: "
   << hist.getOverflowSumWeights() << endl;
```

The `getPDF(x)` and `getCDF(x)` member functions return the value of the Probability Density Function and the Cumulated Density Function at a given *x*, respectively.

Note that bins may not be immediately available during observation collection, because some histogram strategies use precollection to gather information about the distribution before setting up the bins. Use `binsAlreadySetUp()` to figure out whether bins are set up already. Setting up the bins can be forced with the `setupBins()` method.

Setting Up and Managing the Bins

The cHistogram class has several methods for creating and manipulating bins. These methods are primarily intended to be called from strategy classes, but are also useful if you want to manage the bins manually, i.e. without a strategy class.

For setting up the bins, you can either use createUniformBins() with the range (lo, hi) and the step size as parameters, or specify all bin edges explicitly in a vector of doubles to setBinEdges().

When the bins have already been set up, the histogram can be extended with new bins down or up using the prependBins() and appendBins() methods that take a list of new bin edges to add. There is also an extendBinsTo() method that extends the histogram with equal-sized bins at either end to make sure that a supplied value falls into the histogram range. Of course, extending the histogram is only possible if there are no outliers in that direction. (The positions of the outliers is not preserved, so it is not known how many would fall in each of the newly created bins.)

If the histogram has too many bins, adjacent ones (pairs, triplets, or groups of size n) can be merged, using the mergeBins() method.

Example code which sets up a histogram with uniform bins:

```
cHistogram hist("queueLength", nullptr); // create w/o strategy object
hist.createUniformBins(0, 100, 10); // 10 bins over (0,100)
```

The following code achieves the same, but uses setBinEdges():

```
std::vector<double> edges = {0,10,20,30,40,50,60,70,80,90,100}; // C++11
cHistogram hist("queueLength", nullptr);
hist.setBinEdges(edges);
```

Strategy Concept

Histogram strategies subclass from cIHistogramStrategy, and are responsible for setting up and managing the bins.

A cHistogram is created with a cDefaultHistogramStrategy by default, which works well in most cases. Other cHistogram constructors allow passing in an arbitrary histogram strategy.

The collect() and collectWeighted() methods of a cHistogram delegate to similar methods of the strategy object, which in turn decides when and how to set up the bins, and how to manage the bins later. (Setting up the bins may be postponed until a few observations have been collected, in order to gather more information for it.) The histogram strategy uses public histogram methods like createUniformBins() to create and manage the bins.

Available Histogram Strategies

The following histogram strategy classes exist.

cFixedRangeHistogramStrategy sets up uniform bins over a predetermined interval. The number of bins and the histogram mode (integers or reals) also need to be configured. This strategy does not use precollection, as all input for setting up the bins must be explicitly provided by the user.

cDefaultHistogramStrategy is used by the default setup of cHistogram. This strategy uses precollection to gather input information about the distribution before setting up the

bins. Precollection is used to determine the initial histogram range and the histogram mode (integers vs. reals). In integers mode, bin edges will be whole numbers.

To keep up with distributions that change over time, this histogram strategy can auto-extend the histogram range by adding new bins as needed. It also performs bin merging when necessary, to keep the number of bins reasonably low.

`cAutoRangeHistogramStrategy` is a generic, very configurable, precollection-based histogram strategy which creates uniform bins, and creates quality histograms for practical distributions.

Several constraints and hints can be specified for setting up the bins: range lower and/or upper endpoint, bin size, number of bins, mode (integers vs. reals), and whether bin size rounding is to be used.

This histogram strategy can auto-extend the histogram range by adding new bins at either end. One can also set up an upper limit to the number of histogram bins to prevent it from growing indefinitely. Bin merging can also be enabled: it will cause every two (or N) adjacent bins to be merged to reduce the number of bins if their number grows too high.

Random Number Generation from Distributions

The `draw()` member function generates random numbers from the distribution stored by the object:

```
double rnd = histogram.draw();
```

Storing and Loading Distributions

The statistic classes have `loadFromFile()` member functions that read the histogram data from a text file. If you need a custom distribution that cannot be written (or it is inefficient) as a C++ function, you can describe it in histogram form stored in a text file, and use a histogram object with `loadFromFile()`.

You can also use `saveToFile()` that writes out the distribution collected by the histogram object:

```
FILE *f = fopen("histogram.dat","w");
histogram.saveToFile(f); // save the distribution
fclose(f);

cHistogram restored;
FILE *f2 = fopen("histogram.dat","r");
restored.loadFromFile(f2); // load stored distribution
fclose(f2);
```

7.9.3 cPSquare

The `cPSquare` class implements the P^2 algorithm described in [JC85]. P^2 is a heuristic algorithm for dynamic calculation of the median and other quantiles. The estimates are produced dynamically as the observations arrive. The observations are not stored; therefore, the algorithm has a very small and fixed storage requirement regardless of the number of observations. The P^2 algorithm operates by adaptively shifting bin edges as observations arrive.

`cPSquare` only needs the number of cells, for example in the constructor:

```
cPSquare psquare("endToEndDelay", 20);
```

Afterwards, observations can be added and the resulting histogram can be queried with the same `cAbstractHistogram` methods as with `cHistogram`.

7.9.4 cKSplit

Motivation

The *k*-split algorithm is an on-line distribution estimation method. It was designed for on-line result collection in simulation programs. The method was proposed by Varga and Fakhamzadeh in 1997. The primary advantage of *k*-split is that without having to store the observations, it gives a good estimate without requiring a-priori information about the distribution, including the sample size. The *k*-split algorithm can be extended to multi-dimensional distributions, but here we deal with the one-dimensional version only.

The k-split Algorithm

The *k-split* algorithm is an adaptive histogram-type estimate which maintains a good partitioning by doing cell splits. We start out with a histogram range $[x_{lo}, x_{hi})$ with k equal-sized histogram cells with observation counts $n_1, n_2, \cdots n_k$. Each collected observation increments the corresponding observation count. When an observation count n_i reaches a *split threshold*, the cell is split into k smaller, equal-sized cells with observation counts $n_{i,1}, n_{i,2}, \cdots n_{i,k}$ initialized to zero. The n_i observation count is remembered and is called the *mother observation count* to the newly created cells. Further observations may cause cells to be split further (e.g. $n_{i,1,1}, ...n_{i,1,k}$ etc.), thus creating a k-order tree of observation counts where leaves contain live counters that are actually incremented by new observations, and intermediate nodes contain mother observation counts for their children. If an observation falls outside the histogram range, the range is extended in a natural manner by inserting new level(s) at the top of the tree. The fundamental parameter to the algorithm is the split factor k. Experience has shown that $k = 2$ works best.

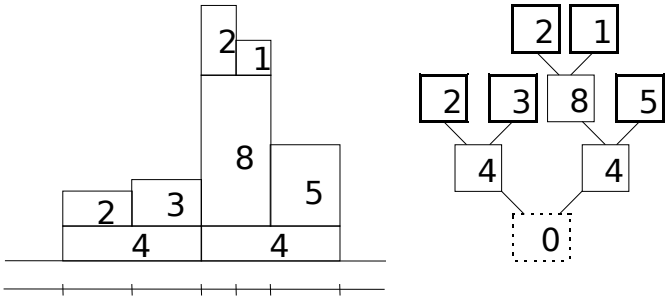


Figure 7.7: Illustration of the k-split algorithm, $k = 2$. The numbers in boxes represent the observation count values

For density estimation, the total number of observations that fell into each cell of the partition has to be determined. For this purpose, mother observations in each internal node of the tree must be distributed among its child cells and propagated up to the leaves.

Let $n_{...,i}$ be the (mother) observation count for a cell, $s_{...,i}$ be the total observation count in a cell $n_{...,i}$ plus the observation counts in all its sub-, sub-sub-, etc. cells), and $m_{...,i}$ the mother observations propagated to the cell. We are interested in the $\tilde{n}_{...,i} = n_{...,i} + m_{...,i}$ estimated amount of observations in the tree nodes, especially in the leaves. In other words, if we have $\tilde{n}_{...,i}$ estimated observation amount in a cell, how to divide it to obtain $m_{...,i,1}, m_{...,i,2} \cdots m_{...,i,k}$ that can be propagated to child cells. Naturally, $m_{...,i,1} + m_{...,i,2} + \cdots + m_{...,i,k} = \tilde{n}_{...,i}$.

Two natural distribution methods are even distribution (when $m_{...,i,1} = m_{...,i,2} = \cdots = m_{...,i,k}$) and proportional distribution (when $m_{...,i,1} : m_{...,i,2} : \cdots : m_{...,i,k} = s_{...,i,1} : s_{...,i,2} : \cdots : s_{...,i,k}$). Even distribution is optimal when the $s_{...,i,j}$ values are very small, and proportional distribution is good when the $s_{...,i,j}$ values are large compared to $m_{...,i,j}$. In practice, a linear combination of them seems appropriate, where $\lambda = 0$ means even and $\lambda = 1$ means proportional distribution:

$m_{...,i,j} = (1-\lambda)\tilde{n}_{...,i}/k + \lambda\tilde{n}_{...,i}s_{...,i,j}/s_{...,i}$ where $\lambda \in [0,1]$

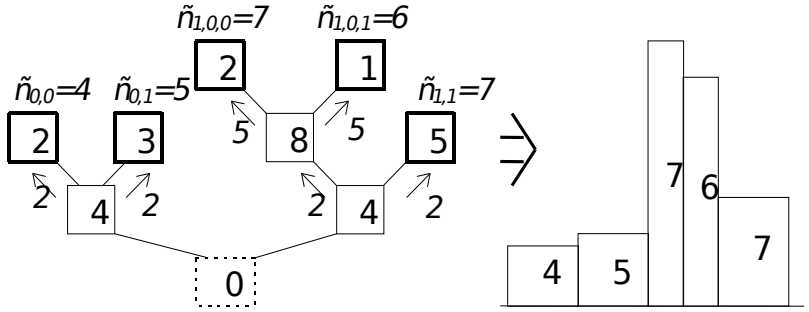


Figure 7.8: Density estimation from the k-split cell tree. We assume $\lambda = 0$, i.e. we distribute mother observations evenly.

Note that while $n_{...,i}$ are integers, $m_{...,i}$ and thus $\tilde{n}_{...,i}$ are typically real numbers. The histogram estimate calculated from *k*-split is not exact, because the frequency counts calculated in the above manner contain a degree of estimation themselves. This introduces a certain *cell division error*; the λ parameter should be selected so that it minimizes that error. It has been shown that the cell division error can be reduced to a more-than-acceptable small value.
Strictly speaking, the *k*-split algorithm is semi-online, because its needs some observations to set up the initial histogram range. Because of the range extension and cell split capabilities, the algorithm is not very sensitive to the choice of the initial range, so very few observations are sufficient for range estimation (say $N_{pre} = 10$). Thus we can regard *k*-split as an on-line method.

K-split can also be used in semi-online mode, when the algorithm is only used to create an optimal partition from a larger number of N_{pre} observations. When the partition has been created, the observation counts are cleared and the N_{pre} observations are fed into *k*-split once again. This way all mother (non-leaf) observation counts will be zero and the cell division error is eliminated. It has been shown that the partition created by *k*-split can be better than both the equi-distant and the equal-frequency partition.

OMNEST contains an implementation of the *k*-split algorithm, the `cKSplit` class.

The cKSplit Class

The `cKSplit` class is an implementation of the *k-split* method. It is a subclass of `cAbstractHistogram`, so configuring, adding observations and querying histogram cells is done the same way as with other histogram classes.

Specific member functions allow one to fine-tune the *k*-split algorithm. `setCritFunc()` and `setDivFunc()` let one replace the split criteria and the cell division function, respectively. `setRangeExtension()` lets one enable/disable range extension. (If range extension is disabled, out-of-range observations will simply be counted as underflows or overflows.)

The class also allows one to access the *k*-split data structure, directly, via methods like `getTreeDepth()`, `getRootGrid()`, `getGrid(i)`, and others.

7.10 Recording Simulation Results

7.10.1 Output Vectors: cOutVector

Objects of type `cOutVector` are responsible for writing time series data (referred to as *output vectors*) to a file. The `record()` method is used to output a value (or a value pair) with a timestamp. The object name will serve as the name of the output vector.

The vector name can be passed in the constructor,

```
cOutVector responseTimeVec("response time");
```

but in the usual arrangement you'd make the `cOutVector` a member of the module class and set the name in `initialize()`. You'd record values from `handleMessage()` or from a function called from `handleMessage()`.

The following example is a `Sink` module which records the lifetime of every message that arrives to it.

```
class Sink : public cSimpleModule
{
  protected:
    cOutVector endToEndDelayVec;

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(Sink);

void Sink::initialize()
{
    endToEndDelayVec.setName("End-to-End Delay");
}

void Sink::handleMessage(cMessage *msg)
{
    simtime_t eed = simTime() - msg->getCreationTime();
    endToEndDelayVec.record(eed);
    delete msg;
}
```

There is also a `recordWithTimestamp()` method, to make it possible to record values into output vectors with a timestamp other than `simTime()`. Increasing timestamp order is still enforced though.

All cOutVector objects write to a single *output vector file* that has a file extension .vec. [5] The format and processing of output vector files is described in section 12.

You can configure output vectors from omnetpp.ini: you can disable individual vectors, or limit recording to certain simulation time intervals (see sections 12.2.2, 12.2.5).

If the output vector object is disabled or the simulation time is outside the specified interval, record() doesn't write anything to the output file. However, if you have a Qtenv inspector window open for the output vector object, the values will be displayed there, regardless of the state of the output vector object.

7.10.2 Output Scalars

While output vectors are to record time series data and thus they typically record a large volume of data during a simulation run, output scalars are supposed to record a single value per simulation run. You can use output scalars

- to record summary data at the end of the simulation run
- to do several runs with different parameter settings/random seed and determine the dependence of some measures on the parameter settings. For example, multiple runs and output scalars are the way to produce *Throughput vs. Offered Load* plots.

Output scalars are recorded with the record() method of cSimpleModule, and you will usually want to insert this code into the finish() function. An example:

```
void Transmitter::finish()
{
    double avgThroughput = totalBits / simTime();
    recordScalar("Average throughput", avgThroughput);
}
```

You can record whole statistic objects by calling their record() methods, declared as part of cStatistic. In the following example we create a Sink module which calculates the mean, standard deviation, minimum and maximum values of a variable, and records them at the end of the simulation.

```
class Sink : public cSimpleModule
{
  protected:
    cStdDev eedStats;

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
};

Define_Module(Sink);

void Sink::initialize()
{
```

[5] A .vci file is also created, but it is just an index for the .vec file and does not contain any new information. The IDE re-creates the .vci file if it gets lost.

```
    eedStats.setName("End-to-End Delay");
}

void Sink::handleMessage(cMessage *msg)
{
    simtime_t eed = simTime() - msg->getCreationTime();
    eedStats.collect(eed);
    delete msg;
}

void Sink::finish()
{
    recordScalar("Simulation duration", simTime());
    eedStats.record();
}
```

The above calls record the data into an *output scalar file*, a line-oriented text file that has the file extension `.sca`. The format and processing of output vector files is described in chapter 12.

7.11 Watches and Snapshots

7.11.1 Basic Watches

Unfortunately, variables of type `int`, `long`, `double` do not show up by default in Qtenv; neither do STL classes (`std::string`, `std::vector`, etc.) or your own structs and classes. This is because the simulation kernel, being a library, knows nothing about types and variables in your source code.

OMNEST provides `WATCH()` and a set of other macros to allow variables to be inspectable in Qtenv and to be output into the snapshot file. `WATCH()` macros are usually placed into `initialize()` (to watch instance variables) or to the top of the `activity()` function (to watch its local variables); the point being that they should only be executed once.

```
long packetsSent;
double idleTime;

WATCH(packetsSent);
WATCH(idleTime);
```

Of course, members of classes and structs can also be watched:

```
WATCH(config.maxRetries);
```

The Qtenv runtime environment lets you inspect and also change the values of inspected variables.

The `WATCH()` macro can be used with any type that has a stream output operator (`operator<<`) defined. By default, this includes all primitive types and `std::string`, but since you can write `operator<<` for your classes/structs and basically any type, `WATCH()` can be used with anything. The only limitation is that since the output should more or less fit on single line, the amount of information that can be conveniently displayed is limited.

An example stream output operator:

```
std::ostream& operator<<(std::ostream& os, const ClientInfo& cli)
{
    os << "addr=" << cli.clientAddr << "  port=" << cli.clientPort; // no endl!
    return os;
}
```

And the `WATCH()` line:

```
WATCH(currentClientInfo);
```

7.11.2 Read-write Watches

Watches for primitive types and `std::string` allow for changing the value from the GUI as well, but for other types you need to explicitly add support for that. What you need to do is define a stream input operator (`operator»`) and use the `WATCH_RW()` macro instead of `WATCH()`.

The stream input operator:

```
std::ostream& operator>>(std::istream& is, ClientInfo& cli)
{
    // read a line from "is" and parse its contents into "cli"
    return is;
}
```

And the `WATCH_RW()` line:

```
WATCH_RW(currentClientInfo);
```

7.11.3 Structured Watches

`WATCH()` and `WATCH_RW()` are basic watches; they allow one line of (unstructured) text to be displayed. However, if you have a data structure generated from message definitions (see Chapter 5), then there is a better approach. The message compiler automatically generates meta-information describing individual fields of the class or struct, which makes it possible to display the contents on field level.

The `WATCH` macros to be used for this purpose are `WATCH_OBJ()` and `WATCH_PTR()`. Both expect the object to be subclassed from `cObject`; `WATCH_OBJ()` expects a reference to such class, and `WATCH_PTR()` expects a pointer variable.

```
ExtensionHeader hdr;
ExtensionHeader *hdrPtr;
...
WATCH_OBJ(hdr);
WATCH_PTR(hdrPtr);
```

CAUTION: With `WATCH_PTR()`, the pointer variable must point to a valid object or be `nullptr` at all times, otherwise the GUI may crash while trying to display the object. This practically means that the pointer should be initialized to `nullptr` even if not used, and should be set to `nullptr` when the object to which it points is deleted.

```
delete watchedPtr;
watchedPtr = nullptr;  // set to nullptr when object gets deleted
```

7.11.4 STL Watches

The standard C++ container classes (`vector`, `map`, `set`, etc) also have structured watches, available via the following macros:

`WATCH_VECTOR()`, `WATCH_PTRVECTOR()`, `WATCH_LIST()`, `WATCH_PTRLIST()`, `WATCH_SET()`, `WATCH_PTRSE`
`WATCH_MAP()`, `WATCH_PTRMAP()`.

The `PTR`-less versions expect the data items ("T") to have stream output operators (`operator «`), because that is how they will display them. The `PTR` versions assume that data items are pointers to some type which has `operator «`. `WATCH_PTRMAP()` assumes that only the value type ("second") is a pointer, the key type ("first") is not. (If you happen to use pointers as key, then define `operator «` for the pointer type itself.)

Examples:

```
std::vector<int> intvec;
WATCH_VECTOR(intvec);

std::map<std::string,Command*> commandMap;
WATCH_PTRMAP(commandMap);
```

7.11.5 Snapshots

The `snapshot()` function outputs textual information about all or selected objects of the simulation (including the objects created in module functions by the user) into the snapshot file.

```
bool snapshot(cObject *obj=nullptr, const char *label=nullptr);
```

The function can be called from module functions, like this:

```
snapshot();      // dump the network
snapshot(this); // dump this simple module and all its objects
snapshot(getSimulation()->getFES()); // dump the future events set
```

`snapshot()` will append to the end of the snapshot file. The snapshot file name has an extension of `.sna`.

The snapshot file output is detailed enough to be used for debugging the simulation: by regularly calling `snapshot()`, one can trace how the values of variables, objects changed over the simulation. The arguments: label is a string that will appear in the output file; obj is the object whose inside is of interest. By default, the whole simulation (all modules etc) will be written out.

If you run the simulation with Qtenv, you can also create a snapshot from the menu.

An example snapshot file (some abbreviations have been applied):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<snapshot object="simulation" label="Long queue" simtime="9.038229311343"
network="FifoNet">
  <object class="cSimulation" fullpath="simulation">
    <info></info>
    <object class="cModule" fullpath="FifoNet">
      <info>id=1</info>
      <object class="fifo::Source" fullpath="FifoNet.gen">
```

```
        <info>id=2</info>
        <object class="cPar" fullpath="FifoNet.gen.sendIaTime">
          <info>exponential(0.01s)</info>
        </object>
        <object class="cGate" fullpath="FifoNet.gen.out">
          <info>--&gt; fifo.in</info>
        </object>
      </object>
      <object class="fifo::Fifo" fullpath="FifoNet.fifo">
        <info>id=3</info>
        <object class="cPar" fullpath="FifoNet.fifo.serviceTime">
          <info>0.01</info>
        </object>
        <object class="cGate" fullpath="FifoNet.fifo.in">
          <info>&lt;-- gen.out</info>
        </object>
        <object class="cGate" fullpath="FifoNet.fifo.out">
          <info>--&gt; sink.in</info>
        </object>
        <object class="cQueue" fullpath="FifoNet.fifo.queue">
          <info>length=13</info>
          <object class="cMessage" fullpath="FifoNet.fifo.queue.job">
            <info>src=FifoNet.gen (id=2)  dest=FifoNet.fifo (id=3)</info>
          </object>
          <object class="cMessage" fullpath="FifoNet.fifo.queue.job">
            <info>src=FifoNet.gen (id=2)  dest=FifoNet.fifo (id=3)</info>
          </object>
        </object>
      <object class="fifo::Sink" fullpath="FifoNet.sink">
        <info>id=4</info>
        <object class="cGate" fullpath="FifoNet.sink.in">
          <info>&lt;-- fifo.out</info>
        </object>
      </object>
    </object>
    <object class="cEventHeap" fullpath="simulation.scheduled-events">
      <info>length=3</info>
      <object class="cMessage" fullpath="simulation.scheduled-events.job">
        <info>src=FifoNet.fifo (id=3)  dest=FifoNet.sink (id=4)</info>
      </object>
      <object class="cMessage" fullpath="...sendMessageEvent">
        <info>at T=9.0464.., in dt=0.00817..; selfmsg for FifoNet.gen (id=2)</info
      </object>
      <object class="cMessage" fullpath="...end-service">
        <info>at T=9.0482.., in dt=0.01; selfmsg for FifoNet.fifo (id=3)</info>
      </object>
    </object>
  </object>
</snapshot>
```

7.11.6 Getting Coroutine Stack Usage

It is important to choose the correct stack size for modules. If the stack is too large, it unnecessarily consumes memory; if it is too small, stack violation occurs.

OMNEST contains a mechanism that detects stack overflows. It checks the intactness of a predefined byte pattern (`0xdeadbeef`) at the stack boundary, and reports "stack violation" if it was overwritten. The mechanism usually works fine, but occasionally it can be fooled by large – and not fully used – local variables (e.g. char buffer[256]): if the byte pattern happens to fall in the middle of such a local variable, it may be preserved intact and OMNEST does not detect the stack violation.

To be able to make a good guess about stack size, you can use the `getStackUsage()` call which tells you how much stack the module actually uses. It is most conveniently called from `finish()`:

```
void FooModule::finish()
{
    EV << getStackUsage() << " bytes of stack used\n";
}
```

The value includes the extra stack added by the user interface library (see *extraStackforEnvir* in envir/envirbase.h), which is currently 8K for Cmdenv and at least 80K for Qtenv. [6]

`getStackUsage()` also works by checking the existence of predefined byte patterns in the stack area, so it is also subject to the above effect with local variables.

7.12 Defining New NED Functions

It is possible to extend the NED language with new functions beyond the built-in ones. New functions are implemented in C++, and then compiled into the simulation model. When a simulation program starts up, the new functions are registered in the NED runtime, and become available for use in NED and ini files.

There are two methods to define NED functions. The `Define_NED_Function()` macro is the more flexible, preferred method of the two. `Define_NED_Math_Function()` is the older one, and it supports only certain cases. Both macros have several variants. [7]

7.12.1 Define_NED_Function()

The `Define_NED_Function()` macro lets you define new functions that can accept arguments of various data types (`bool`, `double`, `string`, etc.), supports optional arguments and also variable argument lists (variadic functions).

The macro has two variants:

```
Define_NED_Function(FUNCTION,SIGNATURE);
Define_NED_Function2(FUNCTION,SIGNATURE,CATEGORY,DESCRIPTION);
```

The two variants are basically equivalent; the only difference is that the second one allows you to specify two more parameters, `CATEGORY` and `DESCRIPTION`. These two parameters expect human-readable strings that are displayed when listing the available NED functions.

[6] The actual value is platform-dependent.

[7] Before OMNEST 4.2, `Define_NED_Math_Function()` was called `Define_Function()`.

The common parameters, FUNCTION and SIGNATURE are the important ones. FUNCTION is the name of (or pointer to) the C++ function that implements the NED function, and SIGNATURE is the function signature as a string; it defines the name, argument types and return type of the NED function.

You can list the available NED functions by running opp_run or any simulation executable with the -h nedfunctions option. The result will be similar to what you can see in Appendix D.

```
$ opp_run -h nedfunctions
OMNeT++ Discrete Event Simulation...
Functions that can be used in NED expressions and in omnetpp.ini:

 Category "conversion":
  double : double double(any x)
    Converts x to double, and returns the result. A boolean argument becomes
    0 or 1; a string is interpreted as number; an XML argument causes an error.
  ...
```

Seeing the above output, it should now be obvious what the CATEGORY and DESCRIPTION macro arguments are for. OMNEST uses the following category names: "conversion", "math", "misc", "ned", "random/continuous", "random/discrete", "strings", "units", "xml". You can use these category names for your own functions as well, when appropriate.

The Signature

The signature string has the following syntax:

returntype functionname(argtype1 argname1, argtype2 argname2, ...)

The *functionname* part defines the name of the NED function, and it must meet the syntactical requirements for NED identifiers (start with a letter or underscore, not be a reserved NED keyword, etc.)

The argument types and return type can be one of the following: **bool**, **int** (maps to C/C++ long), **double**, **quantity**, **string**, **xml** or **any**; that is, any NED parameter type plus **quantity** and **any**. **quantity** means *double with an optional measurement unit* (**double** and **int** only accept dimensionless numbers), and **any** stands for any type. The argument names are presently ignored.

To make arguments optional, append a question mark to the argument name. Like in C++, optional arguments may only occur at the end of the argument list, i.e. all arguments after an optional argument must also be optional. The signature string does not have syntax for supplying default values for optional arguments; that is, default values have to be built into the C++ code that implements the NED function. To let the NED function accept any number of additional arguments of arbitrary types, add an ellipsis (...) to the signature as the last argument.

Some examples:

```
"int factorial(int n)"
"bool isprime(int n)"
"double sin(double x)"
"string repeat(string what, int times)"
"quantity uniform(quantity a, quantity b, long rng?)"
```

```
"any choose(int index, ...)"
```

The first three examples define NED functions with the names `factorial`, `isprime` and `sin`, with the obvious meanings. The fourth example can be the signature for a function that repeats a string n times, and returns the concatenated result. The fifth example is the signature of the existing `uniform()` NED function; it accepts numbers both with and without measurement units (of course, when invoked with measurement units, both `a` and `b` must have one, and the two must be compatible – this should be checked by the C++ implementation). `uniform()` also accepts an optional third argument, an RNG index. The sixth example can be the signature of a `choose()` NED function that accepts an integer plus any number of additional arguments of any type, and returns the *index*th one among them.

Implementing the NED Function

The C++ function that implements the NED function must have one of the following signatures, as defined by the `NedFunction` and `NedFunctionExt` typedefs:

```
cValue function(cComponent *context, cValue argv[], int argc);
cValue function(cExpression::Context *context, cValue argv[], int argc);
```

As you can see, the function accepts an array of `cValue` objects, and returns a `cValue`; the *argc-argv* style argument list should be familiar to you from the declaration of the C/C++ `main()` function. `cValue` is a class that is used during the evaluation of NED expressions, and represents a value together with its type. The `context` argument contains the module or channel in the context of which the NED expression is being evaluated; it is useful for implementing NED functions like `getParentModuleIndex()`.

The function implementation does not need to worry too much about checking the number and types of the incoming arguments, because the NED expression evaluator already does that: inside the function you can be sure that the number and types of arguments correspond to the function signature string. Thus, `argc` is mostly useful only if you have optional arguments or a variable argument list. The NED expression evaluator also checks that the value you return from the function corresponds to the signature.

`cValue` can store all the needed data types (`bool`, `double`, `string`, etc.), and is equipped with the functions necessary to conveniently read and manipulate the stored value. The value can be read via functions like `boolValue()`, `intValue()`, `doubleValue()`, `stringValue()` (returns `const char *`), `stdstringValue()` (returns `const std::string&`) and `xmlValue()` (returns `cXMLElement*`), or by simply casting the object to the desired data type, making use of the provided typecast operators. Invoking a getter or typecast operator that does not match the stored data type will result in a runtime error. For setting the stored value, `cValue` provides a number of overloaded `set()` functions, assignment operators and constructors.

Further `cValue` member functions provide access to the stored data type; yet other functions are associated with handling quantities, i.e. doubles with measurement units. There are member functions for getting and setting the number part and the measurement unit part separately; for setting the two components together; and for performing unit conversion.

Equipped with the above information, we can already write a simple NED function that returns the length of a string:

```
static cValue ned_strlen(cComponent *context, cValue argv[], int argc)
{
    return (long)argv[0].stdstringValue().size();
}
```

```
Define_NED_Function(ned_strlen, "int length(string s)");
```

Note that since `Define_NED_Function()` expects the C++ function to be already declared, we place the function implementation in front of the `Define_NED_Function()` line. We also declare the function to be `static`, because its name doesn't need to be visible for the linker. In the function body, we use `std::string`'s `size()` method to obtain the length of the string, and cast the result to `long`; the C++ compiler will convert that into a `cValue` using `cValue`'s `long` constructor. Note that the **`int`** keyword in the signature maps to the C++ type `long`.

The following example defines a `choose()` NED function that returns its kth argument that follows the `index` (k) argument.

```
static cValue ned_choose(cComponent *context, cValue argv[], int argc)
{
    int index = (int)argv[0];
    if (index < 0 || index >= argc-1)
        throw cRuntimeError("choose(): index %d is out of range", index);
    return argv[index+1];
}

Define_NED_Function(ned_choose, "any choose(int index, ...)");
```

Here, the value of `argv[0]` is read using the typecast operator that maps to `intValue()`. (Note that if the value of the `index` argument does not fit into an `int`, the conversion will result in data loss!) The code also shows how to report errors (by throwing a `cRuntimeError`.)

The third example shows how the built-in `uniform()` NED function could be reimplemented by the user:

```
static cValue ned_uniform(cComponent *context, cValue argv[], int argc)
{
    int rng = argc==3 ? (int)argv[2] : 0;
    double argv1converted = argv[1].doubleValueInUnit(argv[0].getUnit());
    double result = uniform((double)argv[0], argv1converted, rng);
    return cValue(result, argv[0].getUnit());
    // or: argv[0].setPreservingUnit(result); return argv[0];
}

Define_NED_Function(ned_uniform, "quantity uniform(quantity a, quantity b, int rng
```

The first line of the function body shows how to supply default values for optional arguments; for the `rng` argument in this case. The next line deals with unit conversion. This is necessary because the `a` and `b` arguments are both quantities and may come in with different measurement units. We use the `doubleValueInUnit()` function to obtain the numeric value of `b` in `a`'s measurement unit. If the two units are incompatible or only one of the parameters have a unit, an error will be raised. If neither parameters have a unit, `doubleValueInUnit()` simply returns the stored `double`. Then we call the `uniform()` C++ function to actually generate a random number, and return it in a temporary object with `a`'s measurement unit. Alternatively, we could have overwritten the numeric part of `a` with the result using `setPreservingUnit()`, and returned just that. If there is no measurement unit, `getUnit()` will return `nullptr`, which is understood by both `doubleValueInUnit()` and the `cValue` constructor.

NOTE: Note that it is OK to change the elements of the `argv[]` vector: they will be discarded (popped off the evaluation stack) by the NED expression evaluator anyway when your function returns.

cValue In More Detail

In the previous section we have given an overview and demonstrated the basic use of the `cValue` class; here we go into further details.

The stored data type can be obtained with the `getType()` function. It returns an enum (`cValue::Type`) that has the following values: `UNDEF`, `BOOL`, `INT`, `DOUBLE`, `STRING`, `XML`. `UNDEF` is synonymous with *unset*; the others correspond to data types: `bool`, `int64_t`, `double`, `const char *` (`std::string`), `cXMLElement`. There is no separate `QUANTITY` type: quantities are also represented with the `DOUBLE` type, which has an optional associated measurement unit.

The `getTypeName()` static function returns the string equivalent of a `cValue::Type`. The utility function `isSet()` returns `true` if the type is different from `UNDEF`; `isNumeric()` returns `true` if the type is `INT` or `DOUBLE`.

```
cValue value = 5.0;
cValue::Type type = value.getType(); // ==> DOUBLE
EV << cValue::getTypeName(type) << endl; // ==> "double"
```

We have already seen that the `DOUBLE` type serves both the **`double`** and **`quantity`** types of the NED function signature, by storing an optional measurement unit (a string) in addition to the `double` variable. A `cValue` can be set to a quantity by creating it with a two-argument constructor that accepts a `double` and a `const char *` for unit, or by invoking a similar two-argument `set()` function. The measurement unit can be read with `getUnit()`, and overwritten with `setUnit()`. If you assign a `double` to a `cValue` or invoke the one-argument `set(double)` method on it, that will clear the measurement unit. If you want to overwrite the number part but preserve the original unit, you need to use the `setPreservingUnit(double)` method.

There are several functions that perform unit conversion. The `doubleValueInUnit()` method accepts a measurement unit, and attempts to return the number in that unit. The `convertTo()` method also accepts a measurement unit, and tries to permanently convert the value to that unit; that is, if successful, it changes both the number and the measurement unit part of the object. The `convertUnit()` static `cValue` member function accepts three arguments: a quantity as a `double` and a unit, and a target unit; and returns the number in the target unit. A `parseQuantity()` static member function parses a string that contains a quantity (e.g. `"5min 48s"`), and return both the numeric value and the measurement unit. Another version of `parseQuantity()` tries to return the value in a unit you specify. All functions raise an error if the unit conversion is not possible, e.g. due to incompatible units.

For performance reasons, `setUnit()`, `convertTo()` and all other functions that accept and store a measurement unit will only store the `const char*` pointer, but do *not* copy the string itself. Consequently, the passed measurement unit pointers must stay valid for at least the lifetime of the `cValue` object, or even longer if the same pointer propagates to other `cValue` objects. It is recommended that you only pass pointers that stay valid during the entire simulation. It is safe to use: (1) string constants from the code; (2) unit strings from other `cValue`s; and (3) pooled strings e.g. from a `cStringPool` or from `cValue`'s static `getPooled()` function.

Example code:

```
// manipulating the number and the measurement unit
cValue value(250,"ms");     // initialize to 250ms
value = 300.0;                 // ==> 300 (clears the unit!)
value.set(500,"ms");           // ==> 500ms
value.setUnit("s");            // ==> 500s (overwrites the unit)
value.setPreservingUnit(180); // ==> 180s (overwrites the number)
value.setUnit(nullptr);        // ==> 180 (clears the unit)

// unit conversion
value.set(500, "ms");          // ==> 500ms
value.convertTo("s");          // ==> 0.5s
double us = value.doubleValueInUnit("us"); // ==> 500000 (value is unchanged)
double bps = cValue::convertUnit(128, "kbps", "bps"); // ==> 128000
double ms = cValue::convertUnit("2min 15.1s", "ms"); // ==> 135100

// getting persistent measurement unit strings
const char *unit = argv[0].stringValue(); // cannot be trusted to persist
value.setUnit(cValue::getPooled(unit)); // use a persistent copy for setUnit()
```

7.12.2 Define_NED_Math_Function()

The `Define_NED_Math_Function()` macro lets you register a C/C++ "mathematical" function as a NED function. The registered C/C++ function may take up to four `double` arguments, and must return a `double`; the NED signature will be the same. In other words, functions registered this way cannot accept any NED data type other than `double`; cannot return anything else than `double`; cannot accept or return values with measurement units; cannot have optional arguments or variable argument lists; and are restricted to four arguments at most. In exchange for these restrictions, the C++ implementation of the functions is a lot simpler.

Accepted function signatures for `Define_NED_Math_Function()`:

```
double f();
double f(double);
double f(double, double);
double f(double, double, double);
double f(double, double, double, double);
```

The simulation kernel uses `Define_NED_Math_Function()` to expose commonly used `<math.h>` functions in the NED language. Most `<math.h>` functions (`sin()`, `cos()`, `fabs()`, `fmod()`, etc.) can be directly registered without any need for wrapper code, because their signatures is already one of the accepted ones listed above.

The macro has the following variants:

```
Define_NED_Math_Function(NAME,ARGCOUNT);
Define_NED_Math_Function2(NAME,FUNCTION,ARGCOUNT);
Define_NED_Math_Function3(NAME,ARGCOUNT,CATEGORY,DESCRIPTION);
Define_NED_Math_Function4(NAME,FUNCTION,ARGCOUNT,CATEGORY,DESCRIPTION);
```

All macros accept the `NAME` and `ARGCOUNT` parameters; they are the intended name of the NED function and the number of `double` arguments the function takes (0..3). `NAME` should be provided without quotation marks (they will be added inside the macro.) Two macros also accept a `FUNCTION` parameter, which is the name of (or pointer to) the implementation C/C++

function. The macros that don't have a FUNCTION parameter simply use the NAME parameter for that as well. The last two macros accept CATEGORY and DESCRIPTION, which have exactly the same role as with Define_NED_Function().

Examples:

```
Define_NED_Math_Function3(sin, 1, "math", "Trigonometric function; see <math.h>");
Define_NED_Math_Function3(cos, 1, "math", "Trigonometric function; see <math.h>");
Define_NED_Math_Function3(pow, 2, "math", "Power-of function; see <math.h>");
```

7.13 Deriving New Classes

7.13.1 cObject or Not?

If you plan to implement a completely new class (as opposed to subclassing something already present in OMNEST), you have to ask yourself whether you want the new class to be based on cObject or not. Note that we are *not* saying you should always subclass from cObject. Both solutions have advantages and disadvantages, which you have to consider individually for each class.

cObject already carries (or provides a framework for) significant functionality that is either relevant to your particular purpose or not. Subclassing cObject generally means you have more code to write (as you *have to* redefine certain virtual functions and adhere to conventions) and your class will be a bit more heavy-weight. However, if you need to store your objects in OMNEST objects like cQueue or you want to store OMNEST classes in your object, then you *must* subclass from cObject. [8]

The most significant features of cObject are the name string (which has to be stored somewhere, so it has its overhead) and ownership management (see section 7.14), which also provides advantages at some cost.

As a general rule, small struct-like classes like IPAddress or MACAddress are better *not* subclassed from cObject. If your class has at least one virtual member function, consider subclassing from cObject, which does not impose any extra cost because it doesn't have data members at all, only virtual functions.

7.13.2 cObject Virtual Methods

Most classes in the simulation class library are descendants of cObject. When deriving a new class from cObject or a cObject descendant, one must redefine certain member functions so that objects of the new class can fully co-operate with the simulation library classes. A list of those methods is presented below.

> **NOTE:** You don't need to worry about the length of the list: most functions are not always required to implement. For example, forEachChild() is only important if the new class is a container.

The following methods **must** be implemented:

[8] For simplicity, in these sections "OMNEST object" should be understood as "object of a class subclassed from cObject"

- *Constructor*. At least two constructors should be provided: one that takes the object name string as `const char *` (recommended by convention), and another one with no arguments (must be present). The two are usually implemented as a single method, with `nullptr` as default name string.
- *Copy constructor*, which must have the following signature for a class `X`: `X(const X&)`.
- *Destructor*.
- *Duplication function*, `X *dup() const`. It should create and return an exact duplicate of the object. It is usually a one-line function that delegates to the copy constructor.
- *Assignment operator*, that is, `X& operator=(const X&)` for a class `X`. It should copy the contents of the other object into this one, *except* the name string. See later what to do if the object contains pointers to other objects.

If the new class contains other objects subclassed from `cObject`, either via pointers or as a data member, the following function **should** be implemented:

- *Iteration function*, `void forEachChild(cVisitor *v)`. The implementation should call the function passed for each object it contains via pointer or as a data member; see the API Reference on `cObject` on how to implement `forEachChild()`. `forEachChild()` makes it possible for Qtenv to display the object tree, to perform searches on it, etc. It is also used by `snapshot()` and some other library functions.

Implementation of the following methods is **recommended**:

- *Object info*, `str()`. The `str()` function should return a one-line string describing the object's contents or state. The text returned by `str()` is displayed at several places in Qtenv. [9]
- *Serialization*, `parsimPack()` and `parsimUnpack()` methods. These methods are needed for parallel simulation, if you want objects of this type to be transmitted across partitions.

It is customary to implement the copy constructor and the assignment operator so that they delegate to the same function of the base class, and invoke a common private `copy()` function to copy the local members.

7.13.3 Class Registration

You should also use the `Register_Class()` macro to register the new class. It is used by the `createOne()` factory function, which can create any object given the class name as a string. `createOne()` is used by the Envir library to implement `omnetpp.ini` options such as `rng-class="..."` or `scheduler-class="..."`. (see Chapter 17)

For example, an `omnetpp.ini` entry such as

```
rng-class = "cMersenneTwister"
```

would result in something like the following code to be executed for creating the RNG objects:

```
cRNG *rng = check_and_cast<cRNG*>(createOne("cMersenneTwister"));
```

[9] Until OMNEST version 5.1, `str()` was called `info()`. There was also a `detailedInfo()` method that was removed in the same version for lack of real usefulness.

But for that to work, we needed to have the following line somewhere in the code:

```
Register_Class(cMersenneTwister);
```

`createOne()` is also needed by the parallel distributed simulation feature (Chapter 16) to create blank objects to unmarshal into on the receiving side.

7.13.4 Details

We'll go through the details using an example. We create a new class `NewClass`, redefine all above mentioned `cObject` member functions, and explain the conventions, rules and tips associated with them. To demonstrate as much as possible, the class will contain an `int` data member, dynamically allocated non-`cObject` data (an array of `double`s), an OMNEST object as data member (a `cQueue`), and a dynamically allocated OMNEST object (a `cMessage`).

The class declaration is the following. It contains the declarations of all methods discussed in the previous section.

```
//
// file: NewClass.h
//
#include <omnetpp.h>

class NewClass : public cObject
{
  protected:
    int size;
    double *array;
    cQueue queue;
    cMessage *msg;
    ...
  private:
    void copy(const NewClass& other); // local utility function
  public:
    NewClass(const char *name=nullptr, int d=0);
    NewClass(const NewClass& other);
    virtual ~NewClass();
    virtual NewClass *dup() const;
    NewClass& operator=(const NewClass& other);

    virtual void forEachChild(cVisitor *v);
    virtual std::string str();
};
```

We'll discuss the implementation method by method. Here is the top of the `.cc` file:

```
//
// file: NewClass.cc
//
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include "newclass.h"
```

```
Register_Class(NewClass);

NewClass::NewClass(const char *name, int sz) : cObject(name)
{
    size = sz;
    array = new double[size];
    take(&queue);
    msg = nullptr;
}
```

The constructor (above) calls the base class constructor with the name of the object, then initializes its own data members. You need to call `take()` for `cOwnedObject`-based data members.

```
NewClass::NewClass(const NewClass& other) : cObject(other)
{
    size = -1; // needed by copy()
    array = nullptr;
    msg = nullptr;
    take(&queue);
    copy(other);
}
```

The copy constructor relies on the private `copy()` function. Note that pointer members have to be initialized (to `nullptr` or to an allocated object/memory) before calling the `copy()` function.

You need to call `take()` for `cOwnedObject`-based data members.

```
NewClass::~NewClass()
{
    delete [] array;
    if (msg->getOwner()==this)
        delete msg;
}
```

The destructor should delete all data structures the object allocated. `cOwnedObject`-based objects should *only* be deleted if they are owned by the object – details will be covered in section 7.14.

```
NewClass *NewClass::dup() const
{
    return new NewClass(*this);
}
```

The `dup()` function is usually just one line, like the one above.

```
NewClass& NewClass::operator=(const NewClass& other)
{
    if (&other==this)
        return *this;

    cOwnedObject::operator=(other);
    copy(other);
    return *this;
}
```

The assignment operator (above) first makes sure that will not try to copy the object to itself, because that can be disastrous. If so (that is, `&other==this`), the function returns immediately without doing anything.

The base class part is copied via invoking the assignment operator of the base class. Then the method copies over the local members using the `copy()` private utility function.

```
void NewClass::copy(const NewClass& other)
{
    if (size != other.size) {
        size = other.size;
        delete array;
        array = new double[size];
    }
    for (int i = 0; i < size; i++)
        array[i] = other.array[i];

    queue = other.queue;
    queue.setName(other.queue.getName());

    if (msg && msg->getOwner()==this)
        delete msg;

    if (other.msg && other.msg->getOwner()==const_cast<cMessage*>(&other))
        take(msg = other.msg->dup());
    else
        msg = other.msg;
}
```

Complexity associated with copying and duplicating the object is concentrated in the `copy()` utility function.

Data members are copied in the normal C++ way. If the class contains pointers, you will most probably want to make a deep copy of the data where they point, and not just copy the pointer values.

If the class contains pointers to OMNEST objects, you need to take ownership into account. If the contained object is *not owned* then we assume it is a pointer to an "external" object, consequently we only copy the pointer. If it is *owned*, we duplicate it and become the owner of the new object. Details of ownership management will be covered in section 7.14.

```
void NewClass::forEachChild(cVisitor *v)
{
    v->visit(queue);
    if (msg)
        v->visit(msg);
}
```

The `forEachChild()` function should call `v->visit(obj)` for each `obj` member of the class. See the API Reference for more information about `forEachChild()`.

```
std::string NewClass::str()
{
    std::stringstream out;
    out << "data=" << data << ", array[0]=" << array[0];
    return out.str();
```

```
}
```

The `str()` method should produce a concise, one-line string about the object. You should try not to exceed 40-80 characters, since the string will be shown in tooltips and listboxes.

See the virtual functions of `cObject` and `cOwnedObject` in the class library reference for more information. The sources of the Sim library (`include/`, `src/sim/`) can serve as further examples.

7.14 Object Ownership Management

7.14.1 The Ownership Tree

OMNEST has a built-in ownership management mechanism which is used for sanity checks, and as part of the infrastructure supporting Qtenv inspectors.

Container classes like `cQueue` own the objects inserted into them, but this is not limited to objects inserted into a container: *every `cOwnedObject`-based object has an owner all the time.* From the user's point of view, ownership is managed transparently. For example, when you create a new `cMessage`, it will be owned by the simple module. When you send it, it will first be handed over to (i.e. change ownership to) the FES, and, upon arrival, to the destination simple module. When you encapsulate the message in another one, the encapsulating message will become the owner. When you decapsulate it again, the currently active simple module becomes the owner.

The `getOwner()` method, defined in `cObject`, returns the owner of the object:

```
cOwnedObject *o = msg->getOwner();
EV << "Owner of " << msg->getName() << " is: " <<
   << "(" << o->getClassName() << ") " << o->getFullPath() << endl;
```

The other direction, enumerating the objects owned can be implemented with the `forEachChild()` method by it looping through all contained objects and checking the owner of each object.

Why Do We Need This?

The traditional concept of object ownership is associated with the "right to delete" objects. In addition to that, keeping track of the owner and the list of objects owned also serves other purposes in OMNEST:

- enables methods like `getFullPath()` to be implemented.
- prevents certain types of programming errors, namely, those associated with wrong ownership handling.
- enables Qtenv to display the list of simulation objects present within a simple module. This is extremely useful for finding memory leaks caused by forgetting to delete messages that are no longer needed.

Some examples of programming errors that can be caught by the ownership facility:

- attempts to send a message while it is still in a queue, encapsulated in another message, etc.
- attempts to send/schedule a message while it is still owned by the simulation kernel (i.e. scheduled as a future event)
- attempts to send the very same message object to multiple destinations at the same time (ie. to all connected modules)

For example, the `send()` and `scheduleAt()` functions check that the message being sent/scheduled is owned by the module. If it is not, then it signals a programming error: the message is probably owned by another module (already sent earlier?), or currently scheduled, or inside a queue, a message or some other object – in either case, the module does not have any authority over it. When you get the error message (`"not owner of object"`), you need to carefully examine the error message to determine which object has ownership of the message, and correct the logic that caused the error.

The above errors are easy to make in the code, and if not detected automatically, they could cause random crashes which are usually very difficult to track down. Of course, some errors of the same kind still cannot be detected automatically, like calling member functions of a message object which has been sent to (and so is currently owned by) another module.

7.14.2 Managing Ownership

Ownership is managed transparently for the user, but this mechanism has to be supported by the participating classes themselves. It will be useful to look inside `cQueue` and `cArray`, because they might give you a hint what behavior you need to implement when you want to use non-OMNEST container classes to store messages or other `cOwnedObject`-based objects.

Insertion

`cArray` and `cQueue` have internal data structures (array and linked list) to store the objects which are inserted into them. However, they do *not* necessarily own all of these objects. (Whether they own an object or not can be determined from that object's `getOwner()` pointer.)

The default behaviour of `cQueue` and `cArray` is to take ownership of the objects inserted. This behavior can be changed via the *takeOwnership* flag.

Here is what the *insert* operation of `cQueue` (or `cArray`) does:

- insert the object into the internal array/list data structure
- if the *takeOwnership* flag is true, take ownership of the object, otherwise just leave it with its original owner

The corresponding source code:

```
void cQueue::insert(cOwnedObject *obj)
{
    // insert into queue data structure
    ...

    // take ownership if needed
    if (getTakeOwnership())
```

```
        take(obj);

}
```

Removal

Here is what the *remove* family of operations in `cQueue` (or `cArray`) does:

- remove the object from the internal array/list data structure
- if the object is actually owned by this `cQueue`/`cArray`, release ownership of the object, otherwise just leave it with its current owner

After the object was removed from a `cQueue`/`cArray`, you may further use it, or if it is not needed any more, you can delete it.

The *release ownership* phrase requires further explanation. When you remove an object from a queue or array, the ownership is expected to be transferred to the simple module's local objects list. This is accomplished by the `drop()` function, which transfers the ownership to the object's default owner. `getDefaultOwner()` is a virtual method defined in `cOwnedObject`, and its implementation returns the currently executing simple module's local object list.

As an example, the `remove()` method of `cQueue` is implemented like this: [10]

```
cOwnedObject *cQueue::remove(cOwnedObject *obj)
{
    // remove object from queue data structure
    ...

    // release ownership if needed
    if (obj->getOwner()==this)
        drop(obj);

    return obj;
}
```

Destructor

The concept of `ownership` is that *the owner has the exclusive right and duty to delete the objects it owns*. For example, if you delete a `cQueue` containing `cMessage`s, all messages it contains *and* owns will also be deleted.

The destructor should delete all data structures the object allocated. From the contained objects, only the owned ones are deleted – that is, where `obj->getOwner()==this`.

Object Copying

The ownership mechanism also has to be taken into consideration when a `cArray` or `cQueue` object is duplicated (using `dup()` or the copy constructor.) The duplicate is supposed to

[10] Actual code in `src/sim` is structured somewhat differently, but the meaning is the same.

have the same content as the original; however, the question is whether the contained objects should also be duplicated or only their pointers taken over to the duplicate `cArray` or `cQueue`. A similar question arises when an object is copied using the assignment operator (`operator=()`).

The convention followed by `cArray`/`cQueue` is that only owned objects are copied, and the contained but not owned ones will have their pointers taken over and their original owners left unchanged.

Chapter 8

Graphics and Visualization

8.1 Overview

OMNEST simulations can be run under graphical user interfaces like Qtenv that offer visualization and animation in addition to interactive execution and other features. This chapter deals with model visualization.

OMNEST essentially provides four main tools for defining and enhancing model visualization:

1. *Display strings* is the traditional way. It is a per-component string that encodes how the component (module or channel) will show up in the graphical user interface. Display strings can be specified in NED files, and can also be manipulated programmatically at runtime.

2. *The canvas.* The same user interface area that contains submodules and connections (i.e. the *canvas*) can also display additional graphical elements that OMNEST calls *figures*. Using figures, one can display lines, curves, polygons, images and text items, and anything that can be built by combining them and applying effects like rotation and scaling. Like display strings, figures can also be specified in NED files, but it is generally more useful to create and manipulate them programmatically. Every module has its own default canvas, and extra canvases can also be created at runtime.

3. *3D visualization* of the simulation's virtual world is a third possiblity. OMNEST's 3D visualization capabilities come from the open-source OpenSceneGraph library and its osgEarth extension. These libraries build on top of OpenGL, and beyond basic graphics functionality they also offer high-level capabilities, such as reading 3D model files directly from disk, or displaying maps, 3D terrain or Earth as a planet using online map and satellite imagery data sources.

4. *Support for smooth custom animation* allows models to visualize their operation using sophisticated animations. The key idea is that the simulation model is called back from the runtime GUI (Qtenv) repeatedly at a reasonable "frame rate," allowing it to continually update the canvas (2D) and/or the 3D scene to produce fluid animations.

The following sections will cover the above topics in more detail. But first, let us get acquainted with a new `cModule` virtual method that one can redefine and place visualization-related code into.

8.2 Placement of Visualization Code

Traditionally, when C++ code was needed to enhance visualization, for example to update a displayed status label or to refresh the position of a mobile node, it was embedded in `handleMessage()` functions, enclosed in `if (ev.isGUI())` blocks. This was less than ideal, because the visualization code would run for all events in that module and not just before display updates when it was actually needed. In *Express* mode, for example, Qtenv would only refresh the display once every second or so, with a large number of events processed between updates, so visualization code placed inside `handleMessage()` could potentially waste a significant amount of CPU cycles. Also, visualization code embedded in `handleMessage()` is not suitable for creating smooth animations.

8.2.1 The refreshDisplay() Method

Starting from OMNEST version 5.0, visualization code can be placed into a dedicated method. It is called much more economically, that is, exactly as often as needed.

This method is `refreshDisplay()`, and is declared on `cModule` as:

```
virtual void refreshDisplay() const {}
```

Components that contain visualization-related code are expected to override `refreshDisplay()`, and move visualization code such as display string manipulation, canvas figure maintenance and OSG scene graph updates into it.

When and how is `refreshDisplay()` invoked? Generally, right before the GUI performs a display update. With some additional rules, that boils down to the following:

1. It is invoked only under graphical user interfaces, currently Qtenv. It is never invoked under Cmdenv.
2. When invoked, it will be called on *all* components of the simulation. It does not matter if a module has a graphical inspector open or not. This design decision simplifies the handling of cross-module visualization dependencies. Runtime overhead is still not an issue, because display updates are only done at most a few times per second in *Express* mode, while in other modes, raw event processing performance is of somewhat lesser importance. [1]
3. It is invoked right before display updates. This includes the following: after network setup; in *Step* and *Run* modes, before and after every event; in *Fast* and *Express* modes, after every "batch" of events; every time a new graphical inspector is opened, zoomed, navigated in, or closed; after model data (cPar, cDisplayString values, etc.) is edited, and after finalization.
4. If smooth animation is used, it is invoked continuously with a reasonably high frequency in *Step*, *Run* and *Fast* modes. This can mean anything from many times between processing two consecutive events to not even once until after the processing of a couple of events, depending on the current animation speed and event density.

Here is an example of how one would use it:

[1] At any rate, only a small portion of components are expected to have (nontrivial) `refreshDisplay()` overrides in complex models. If it still becomes too resource-consuming, local caching of related data and the use of a *displayInvalid* flag might help.

```
void FooModule::refreshDisplay() const
{
    // refresh statistics
    char buf[80];
    sprintf(buf, "Sent:%d  Rcvd:%d", numSent, numReceived);
    getDisplayString()->setTagArg("t", 0, buf);

    // update the mobile node's position
    Point pos = ...  // e.g. invoke a computePosition() method
    getDisplayString()->setTagArg("p", 0, pos.x);
    getDisplayString()->setTagArg("p", 1, pos.y);
}
```

One useful accessory to `refreshDisplay()` is the `isExpressMode()` method of `cEnvir`. It returns true if the simulation is running under a GUI in *Express* mode. Visualization code may check this flag and adapt the visualization accordingly. An example:

```
if (getEnvir()->isExpressMode()) {
    // display throughput statistics
}
else {
    // visualize current frame transmission
}
```

8.2.2 Advantages

Overriding `refreshDisplay()` has several advantages over putting the simulation code into `handleMessage()`. The first one is clearly *performance*. When running under Cmdenv, the runtime cost of visualization code is literally zero, and when running in *Express* mode under Qtenv, it is practically zero because the cost of one update is amortized over several hundred thousand or million events.

The second advantage is also very practical: *consistency* of the visualization. If the simulation has cross-module dependencies such that an event processed by one module affects the information displayed by another module, with `handleMessage()`-based visualization the model may have inconsistent visualization until the second module also processes an event and updates its displayed state. With `refreshDisplay()` this does not happen, because all modules are refreshed together.

The third advantage is *separation of concerns*. It is generally not a good idea to intermix simulation logic with visualization code, and `refreshDisplay()` allows one to completely separate the two.

8.2.3 Why is `refreshDisplay()` const?

Code in `refreshDisplay()` should never alter the state of the simulation because that would destroy repeatability, due to the fact that the timing and frequency of `refreshDisplay()` calls is completely unpredictable from the simulation model's point of view. The fact that the method is declared `const` gently encourages this behavior.

If visualization code makes use of internal caches or maintains some other mutable state, such data members can be declared `mutable` to allow `refreshDisplay()` to change them.

8.3 Smooth Animation

8.3.1 Concepts

Support for smooth custom animation allows models to visualize their operation using sophisticated animations. The key idea is that the simulation model is called back from the runtime GUI (Qtenv) repeatedly at a reasonable "frame rate," allowing it to continually update the canvas (2D) and/or the 3D scene to produce fluid animations. Callback means that the `refreshDisplay()` methods of modules and figures are invoked.

`refreshDisplay()` knows the animation position from the simulation time and the *animation time*, a variable also made accessible to the model. If you think about the animation as a movie, animation time is simply the position in seconds in the movie. By default, the movie is played in Qtenv at normal (1x) speed, and then animation time is simply the number of seconds since the start of the movie. The speed control slider in Qtenv's toolbar allows you to play it at higher (2x, 10x, etc.) and lower (0.5x, 0.1x, etc.) speeds; so if you play the movie at 2x speed, animation time will pass twice as fast as real time.

When smooth animation is turned on (more about that later), simulation time progresses in the model (piecewise) linearly. The speed at which the simulation progresses in the movie is called *animation speed*. Sticking to the movie analogy, when the simulation progresses in the movie 100 times faster than animation time, animation speed is 100.

Certain actions take zero simulation time, but we still want to animate them. Examples of such actions are the sending of a message over a zero-delay link, or a visualized C++ method call between two modules. When these animations play out, simulation is paused and simulation time stays constant until the animation is over. Such periods are called *holds*.

8.3.2 Smooth vs. Traditional Animation

Smooth animation is a relatively new feature in OMNEST, and not all simulations need it. Smooth and traditonal, "non-smooth" animation in Qtenv are two distinct modes which operate very differently:

- In **Traditional animation**, simulation events are essentially processed *as fast as possible*, and meanwhile, `refreshDisplay()` is called with some policy (e.g. once before/after each event, or at 1s intervals real-time) to keep the displayed graphics up to date.
- **Smooth animation** is essentially a *scaled realtime simulation*, where `refreshDisplay()` is continually called with a reasonably high frame rate.

The factor that decides which operation mode is active is the *presence of an animation speed*. If there is no animation speed, traditional animation is performed; if there is one, smooth animation is done.

The Qtenv GUI has a dialog (*Animation Parameters*) which displays the current animation speed, among other things. This dialog allows the user to check at any time which operation mode is currently active.[2]

[2] Note that even during traditional animation, some built-in animation effects request animation speeds and holds, so there may be periods when smooth animation is performed.

8.3.3 The Choice of Animation Speed

Different animation speeds may be appropriate for different animation effects. For example, when animating WiFi traffic where various time slots are on the microsecond scale, an animation speed on the order of 10^-5 might be appropriate; when animating the movement of cars or pedestrians, an animation speed of 1 is a reasonable choice. When several animations requiring different animation speeds occur in the same scene, one solution is to animate the scene using the lowest animation speed so that even the fastest actions can be visually followed by the human viewer.

The solution provided by OMNEST for the above problem is the following. Animation speed cannot be controlled explicitly, only requests may be submitted. Several parts of the models may request different animation speeds. The effective animation speed is computed as the minimum of the animation speeds of visible canvases, unless the user interactively overrides it in the UI, for example by imposing a lower or upper limit.

An animation speed requests may be submitted using the `setAnimationSpeed()` method of `cCanvas`.[3] The `setAnimationSpeed()` method takes two arguments: the animation speed value (a `double`) and an object pointer (`cObject*`) that identifies the part of the model that requests it. The second, object parameter is used as a key that allows the request to be updated or withdrawn later. Typically, the pointer of the module that makes the request (i.e. `this`) is used for that purpose. Calling `setAnimationSpeed()` with zero animation speed cancels the request.

An example:

```
cCanvas *canvas = getSystemModule()->getCanvas(); // toplevel canvas
canvas->setAnimationSpeed(2.0, this); // one request
canvas->setAnimationSpeed(1e-6, macModule); // another request
...
canvas->setAnimationSpeed(1.0, this); // overwrite first request
canvas->setAnimationSpeed(0, macModule); // cancel second request
```

In practice, built-in animation effects such as message sending animation also submit their own animation speed requests internally, so they also affect the effective animation speed chosen by Qtenv.

The current effective animation speed can be obtained from the environment of the simulation (`cEnvir`, see chapter 18 for context):

```
double animSpeed = getEnvir()->getAnimationSpeed();
```

Animation time can be accessed like this:

```
double animTime = getEnvir()->getAnimationTime();
```

Animation time starts from zero, and monotonically increases with simulation time and also during "holds".

8.3.4 Holds

As mentioned earlier, a hold interval is an interval when only animation takes place, but simulation time does not progress and no events are processed. Hold intervals are intended for animating actions that take zero simulation time.

[3] The class that represents the canvas for 2D graphics, see 8.6.2 for more info.

A hold can be requested with the `holdSimulationFor()` method of `cCanvas`, which accepts an animation time delta as parameter. If a hold request is issued when there is one already in progress, the current hold will be extended as needed to incorporate the request. A hold request cannot be cancelled or shrunk.

```
cCanvas *canvas = getSystemModule()->getCanvas(); // toplevel canvas
canvas->holdSimulationFor(0.5); // request a 0.5s (animation time) hold
```

When rendering frames in `refreshDisplay()`) during a hold, the code can use animation time to determine the position in the animation. If the code needs to know the animation time elapsed since the start of the hold, it should query and remember the animation time when issuing the hold request.

If the code needs to know the animation time remaining until the end of the hold, it can use the `getRemainingAnimationHoldTime()` method of `cEnvir`. Note that this is not necessarily the time remaining from its own hold request, because other parts of the simulation might extend the hold.

8.3.5 Disabling Built-In Animations

If a model implements such full-blown animations for a compound module that OMNEST's default animations (message sending/method call animations) become a liability, they can be programmatically turned off for that module with `cModule`'s `setBuiltinAnimationsAllowed()` method:

```
// disable animations for the toplevel module
cModule *network = getSimulation()->getSystemModule();
network->setBuiltinAnimationsAllowed(false);
```

8.4 Display Strings

Display strings are compact textual descriptions that specify the arrangement and appearance of the graphical representations of modules and connections in graphical user interfaces (currently Qtenv).

Display strings are usually specified in NED's `@display` property, but it is also possible to modify them programmatically at runtime.

Display strings can be used in the following contexts:

- *submodules* – display strings may contain position, arrangement (for module vectors), icon, icon color, auxiliary icon, status text, communication range (as circle or filled circle), tooltip, etc.
- *compound modules, networks* – display strings can specify background color, border color, border width, background image, scaling, grid, unit of measurement, etc.
- *connections* – display strings can specify positioning, color, line width, line style, text and tooltip
- *messages* – display strings can specify icon, icon color, etc.

8.4.1 Syntax and Placement

Display strings are specified in `@display` properties. The property must contain a single string as value. The string should contain a semicolon-separated list of tags. Each tag consists of a key, an equal sign and a comma-separated list of arguments:

```
@display("p=100,100;b=60,10,rect,blue,black,2")
```

Tag arguments may be omitted both at the end and inside the parameter list. If an argument is omitted, a sensible default value is used. In the following example, the first and second arguments of the `b` tag are omitted.

```
@display("p=100,100;b=,,rect,blue")
```

Display strings can be placed in the *parameters* section of module and channel type definitions, and in submodules and connections. The following NED sample illustrates the placement of display strings in the code:

```
simple Server
{
    parameters:
        @display("i=device/server");
    ...
}

network Example
{
    parameters:
        @display("bgi=maps/europe");
    submodules:
        server: Server {
            @display("p=273,101");
        }
        ...
    connections:
        client1.out --> { @display("ls=red,3"); } --> server.in++;
}
```

8.4.2 Inheritance

At runtime, every module and channel object has one single display string object, which controls its appearance in various contexts. The initial value of this display string object comes from merging the `@display` properties occurring at various places in NED files. This section describes the rules for merging `@display` properties to create the module or channel's display string.

- Derived NED types inherit their display string from their base NED type.
- Submodules inherit their display string from their type.
- Connections inherit their display string from their channel type.

The base NED type's display string is merged into the current display string using the following rules:

1. **Inheriting.** If a tag or tag argument is present in the base display string but not in the current one, it will be added to the result. Example:
   `"i=block/sink"` *(base)* + `"p=20,40;i=,red"` *(current)* → `"p=20,40;i=block/sink,red"`

2. **Overwriting.** If a tag argument is present both in the base and in the current display string, the tag argument in the current display string will win. Example:
   `"b=40,20,oval"` + `"b=,30"` → `"b=40,30,oval"`

3. **Erasing.** If the current display string contains a tag argument with the value "-" (hyphen), that tag argument will be empty in the result. Example:
   `"i=block/sink,red"` + `"i=,-"` → `"i=block/sink"`

The result of merging the `@display` properties will be used to initialize the display string object (`cDisplayString`) of the module or channel. The display string object can then still be modified programmatically at runtime.

> **NOTE:** If a tag argument is empty, the GUI may use a suitable default value. For example, if the border color for a rectangle is not specified in the display string, the GUI may use black. This default value cannot be queried programmatically.

Example of display string inheritance:

```
simple Base {
    @display("i=block/queue"); // use a queue icon in all instances
}

simple Derived extends Base {
    @display("i=,red,60");  // ==> "i=block/queue,red,60"
}

network SimpleQueue {
    submodules:
        submod: Derived {
            @display("i=,yellow,-;p=273,101;r=70");
                     // ==> "i=block/queue,yellow;p=273,101;r=70"
        }
        ...
}
```

8.4.3 Submodule Tags

The following tags of the module display string are in effect in submodule context, that is, when the module is displayed as a submodule of another module:

- `i` – icon
- `is` – icon size
- `i2` – auxiliary or status icon
- `b` – shape (box, oval, etc.)

- `p` – positioning and layout
- `g` – layout group
- `r` – range indicator
- `q` – queue information text
- `t` – text
- `tt` – tooltip

The following sections provide an overview and examples for each tag. More detailed information, such as what each tag argument means, is available in Appendix G.

Icons

By default, modules are displayed with a simple default icon, but OMNEST comes with a large set of categorized icons that one can choose from. To see what icons are available, look into the `images/` folder in the OMNEST installation. The stock icons installed with OMNEST have several size variants. Most of them have very small (vs), small (s), large (l) and very large (vl) versions.

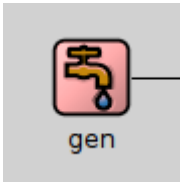
One can specify the icon with the `i` tag. The icon name should be given with the name of the subfolder under `images/`, but without the file name extension. The size may be specified with the icon name suffix (`_s` for very small, `_vl` for very large, etc.), or in a separate `is` tag.

An example that displays the *block/source* in large size:

```
@display("i=block/source;is=l");
```

Icons may also be colorized, which can often be useful. Color can indicate the status or grouping of the module, or simply serve aesthetic purposes. The following example makes the icon 20% red:

```
@display("i=block/source,red,20")
```

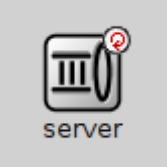


Status Icon

Modules may also display a small auxiliary icon in the top-right corner of the main icon. This icon can be useful for displaying the status of the module, for example, and can be set with the `i2` tag. Icons suitable for use with `i2` are in the `status/` category.

An example:

```
@display("i=block/queue;i2=status/busy")
```



Shapes

To have a simple but resizable representation for a module, one can use the `b` tag to create geometric shapes. Currently, `oval` and `rectangle` are supported.

The following example displays an oval shape of the size 70x30 with a 4-pixel black border and red fill:

```
@display("b=70,30,oval,red,black,4")
```

Positioning

The `p` tag allows one to define the position of a submodule or otherwise affect its placement.

> **NOTE:** If the `p` tag is missing or doesn't specify the position, OMNEST will use a layouting algorithm to place the module automatically. The layouting algorithm is covered in section 8.4.10.

The following example will place the module at the given position:

```
@display("p=50,79");
```

> **NOTE:** Coordinates and distances in `p`, `b` or `r` tags need not be integers. Fractional numbers make sense because runtime GUIs like Qtenv support zooming.

If the submodule is a module vector, one can also specify in the `p` tag how to arrange the elements of the vector. They can be arranged in a row, a column, a matrix or a ring. The rest of the arguments in the `p` tag depend on the layout type:

TODO refine, e.g. list accepted abbreviations for matrix etc; what if x,y are missing; delta args are optional; etc

- Row: `p=`x`,`y`,r,`$deltaX$ (A row of modules with $deltaX$ units between the modules)
- Column: `p=`x`,`y`,c,`$deltaY$ (A column of modules with $deltaY$ units between the modules)
- Matrix: `p=`x`,`y`,m,`$numCols$`,`$deltaX$`,`$deltaY$ (A matrix with $numCols$ columns, with $deltaX$ and $deltaY$ units between rows and columns)

- Ring `p=`x`,`y`,ri,`rx`,`ry (A ring (oval) with rx and ry as the horizontal and vertical radius.)
- Exact (default): `p=`x`,`y`,x,`$deltaX$`,`$deltaY$ (Place each module at $(x+deltaX, y+deltaY)$. The coordinates are often set at runtime.)

A matrix layout for a module vector (note that the first two arguments, x and y are omitted, so the submodule matrix as a whole will be placed by the layouter algorithm):

```
host[20]: Host {
    @display("p=,,m,4,50,50");
}
```

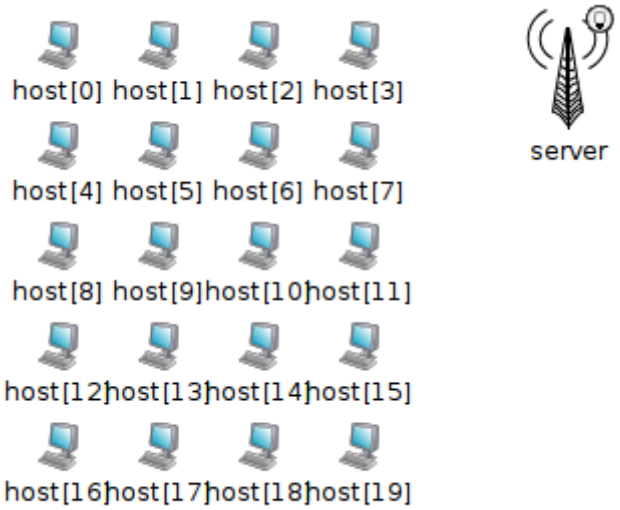


Figure 8.1: Matrix arrangement using the p tag

Layout Group

Layout groups allow modules that are not part of the same submodule vector to be arranged in a row, column, matrix or ring formation as described in the `p` tag's third (and further) parameters.

The `g` tag expects a single string parameter, the group name. All sibling modules that share the same group name are treated for layouting purposes as if they were part of the same submodule vector, the "index" being the order of submodules within their parent.

Wireless Range

In wireless simulations, it is often useful to be able to display a circle or disc around the module to indicate transmission range, reception range, or interference range. This can be done with the `r` tag.

In the following example, the module will have a circle with a 90-unit radius around it as a range indicator:

```
submodules:
    ap: AccessPoint {
        @display("p=50,79;r=90");
    }
```

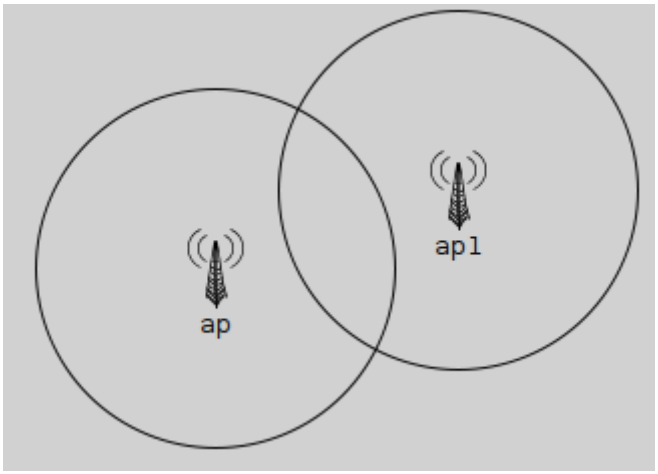


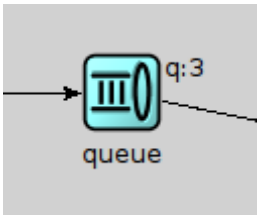
Figure 8.2: Range indicator using the r tag

Queue Length

If a module contains a queue object (`cQueue`), it is possible to let the graphical user interface display the queue length next to the module icon. To achieve that, one needs to specify the queue object's name (the string set via the `setName()` method) in the `q` display string tag. OMNEST finds the queue object by traversing the object tree inside the module.

The following example displays the length of the queue named `"jobQueue"`:

```
@display("q=jobQueue");
```

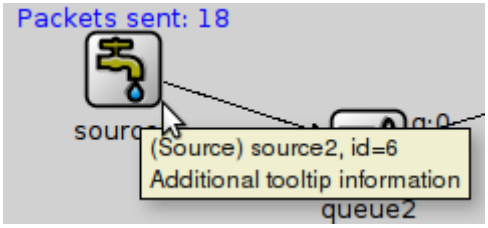


Text and Tooltip

It is possible to have a short text displayed next to or above the module icon or shape using the `t` tag. The tag lets one specify the placement (left, right, above) and the color of the text. To display text in a tooltip, use the `tt` tag.

The following example displays text above the module icon, and also adds tooltip text that can be seen by hovering over the module icon with the mouse.

```
@display("t=Packets sent: 18;tt=Additional tooltip information");
```



NOTE: The `t` and `tt` tags, when set at runtime, can be used to display information about the module's state. The `setTagArg()` method of `cDisplayString` can be used to update the text: `getDisplayString().setTagArg("t", 0, str);`

For a detailed descripton of the display string tags, check Appendix G.

8.4.4 Background Tags

The following tags of the module display string are in effect when the module itself is opened in a GUI. These tags mostly deal with the visual properties of the background rectangle.

- `bgb` – size, color and border of the background rectangle
- `bgi` – background image and its display mode
- `bgtt` – tooltip above the background
- `bgg` – background grid: color, spacing, etc.
- `bgu` – measurement unit of coordinates/distances

In the following example, the background area is defined to be 6000x4500 units, and the map of Europe is used as a background, stretched to fill the whole area. A grid is also drawn, with 1000 units between major ticks, and 2 minor ticks per major tick.

```
network EuropePlayground
{
    @display("bgb=6000,4500;bgi=maps/europe,s;bgg=1000,2,grey95;bgu=km");
```

The `bgu` tag deserves special attention. It does not affect the visual appearance, but instead it is a hint for model code on how to interpret coordinates and distances in this compound module. The above example specifies `bgu=km`, which means that if the model attaches physical meaning to coordinates and distances, then those numbers should be interpreted as kilometers.

More detailed information, such as what each tag argument means, is available in Appendix G.

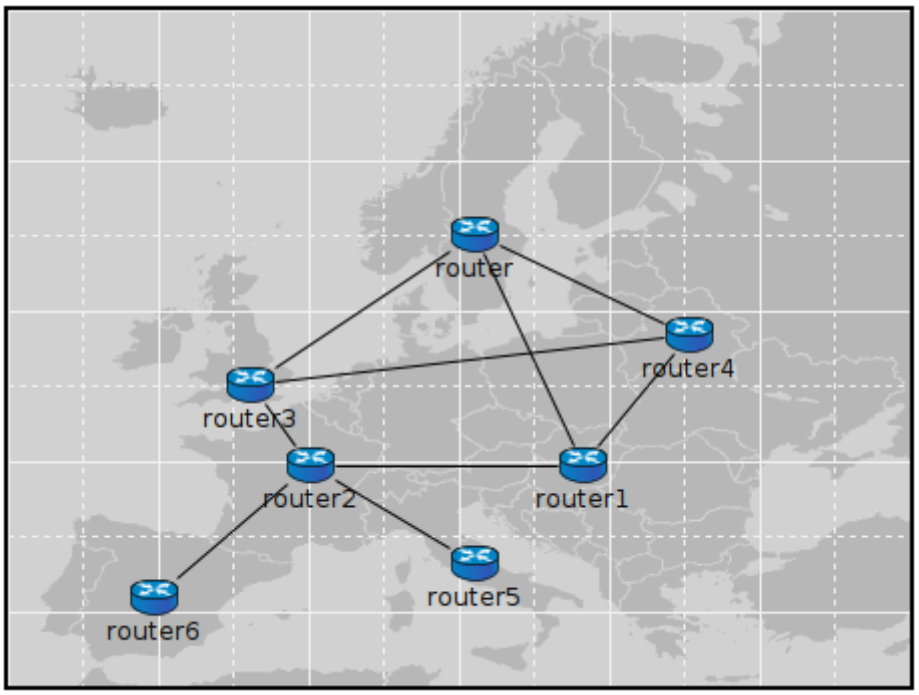


Figure 8.3: Background image and grid

8.4.5 Connection Display Strings

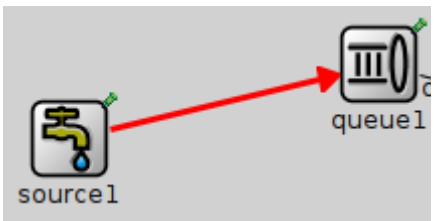
Connections may also have display strings. Connections inherit the display string property from their channel types, in the same way as submodules inherit theirs from module types. The default display strings are empty.

Connections support the following tags:

- `ls` – line style and color
- `t` – text
- `tt` – tooltip
- `m` – orientation and positioning

Example of a thick, red connection:

```
source1.out --> { @display("ls=red,3"); } --> queue1.in++;
```



NOTE: To hide a connection, specify zero line width in the display string: `"ls=,0"`.

More detailed information, such as what each tag argument means, is available in Appendix G.

8.4.6 Message Display Strings

Message display strings affect how messages are shown during animation. By default, they are displayed as a small filled circle, in one of 8 basic colors (the color is determined as *message kind modulo 8*), and with the message class and/or name displayed under it. The latter is configurable in the Preferences dialog of Qtenv, and message kind dependent coloring can also be turned off there.

How to Specify

Message objects do not store a display string by default. Instead, `cMessage` defines a virtual `getDisplayString()` method that one can override in subclasses to return an arbitrary string. The following example adds a display string to a new message class:

```
class Job : public cMessage
{
  public:
    const char *getDisplayString() const {return "i=msg/packet;is=vs";}
    //...
};
```

Since message classes are often defined in `msg` files (see chapter 6), it is often convenient to let the message compiler generate the `getDisplayString()` method. To achieve that, add a string field named `displayString` with an initializer to the message definition. The message compiler will generate `setDisplayString()` and `getDisplayString()` methods into the new class, and also set the initial value in the constructor.

An example message file:

```
message Job
{
    string displayString = "i=msg/package_s,kind";
    //...
}
```

Tags

The following tags can be used in message display strings:

- `b` – shape, color
- `i` – icon
- `is` – icon size

NOTE: In message display strings, `kind` is accepted as a special color name. It will cause the color to be derived from *message kind* field in the message.

The following example displays a small red box icon:

```
@display("i=msg/box,red;is=s");
```

The next one displays a 15x15 rectangle, with while fill, and with a border color dependent on the message kind:

```
@display("b=15,15,rect,white,kind,5");
```

More detailed information, such as what each tag argument means, is available in Appendix G.

8.4.7 Parameter Substitution

Parameters of the module or channel containing the display string can be substituted into the display string with the `$parameterName` notation:

Example:

```
simple MobileNode
{
    parameters:
        double xpos;
        double ypos;
        string fillColor;
        // get the values from the module parameters xpos,ypos,fillcolor
        @display("p=$xpos,$ypos;b=60,10,rect,$fillColor,black,2");
}
```

8.4.8 Colors

Color Names

A color may be given in several forms. One is English names: `blue`, `lightgrey`, `wheat`, etc.; the list includes all standard SVG color names.

Another acceptable form is the HTML RGB syntax: *#rgb* or *#rrggbb*, where *r*,*g*,*b* are hex digits.

It is also possible to specify colors in HSB (hue-saturation-brightness) as *@hhssbb* (with *h*, *s*, *b* being hex digits). HSB makes it easier to scale colors e.g. from white to bright red.

One can produce a transparent background by specifying a hyphen (*"-"*) as background color.

In message display strings, `kind` can also be used as a special color name. It will map message kind to a color. (See the `getKind()` method of `cMessage`.)

Icon Colorization

The `"i="` display string tag allows for colorization of icons. It accepts a target color and a percentage as the degree of colorization. Percentage has no effect if the target color is missing. Brightness of the icon is also affected – to keep the original brightness, specify a color with about 50% brightness (e.g. `#808080` mid-grey, `#008000` mid-green).

Examples:

- "i=device/server,gold" creates a gold server icon
- "i=misc/globe,#808080,100" makes the icon greyscale
- "i=block/queue,white,100" yields a "burnt-in" black-and-white icon

Colorization works with both submodule and message icons.

8.4.9 Icons

The Image Path

In the current OMNEST version, module icons are PNG or GIF files. The icons shipped with OMNEST are in the images/ subdirectory. The IDE and Qtenv need the exact location of this directory to be able to load the icons.

Icons are loaded from all directories in the *image path*, a semicolon-separated list of directories. The default image path is compiled into Qtenv with the value "*<omnetpp>*/images;./images". This works fine (unless the OMNEST installation is moved), and the ./images part also allows icons to be loaded from the images/ subdirectory of the current directory. As users typically run simulation models from the model's directory, this practically means that custom icons placed in the images/ subdirectory of the model's directory are automatically loaded.

The compiled-in image path can be overridden with the OMNETPP_IMAGE_PATH environment variable. The way of setting environment variables is system specific: in Unix, if one is using the bash shell, adding a line

```
export OMNETPP_IMAGE_PATH="$HOME/omnetpp/images;./images"
```

to ~/.bashrc or ~/.bash_profile will do; on Windows, environment variables can be set via the *My Computer –> Properties* dialog.

One can extend the image path from omnetpp.ini with the image-path option, which is prepended to the environment variable's value.

```
[General]
image-path = "/home/you/model-framework/images;/home/you/extra-images"
```

Categorized Icons

Icons are organized into several categories, represented by folders. These categories include:

- abstract/ - symbolic icons for various devices
- background/ - images useful as background, such as terrain map
- block/ - icons for subcomponents (queues, protocols, etc).
- device/ - network device icons: servers, hosts, routers, etc.
- misc/ - node, subnet, cloud, building, town, city, etc.
- msg/ - icons that can be used for messages
- status/ - status icons such as up, down, busy, etc.

Icon names to be used with the `i`, `bgi` and other tags should contain the subfolder (category) name but not the file extension. For example, `/opt/omnetpp/images/block/sink.png` should be referred to as `block/sink`.

Icon Size

Icons come in various sizes: normal, large, small, very small, very large. Sizes are encoded into the icon name's suffix: `_vl`, `_l`, `_s`, `_vs`. In display strings, one can either use the suffix (`"i=device/router_l"`), or the `"is"` (*icon size*) display string tag (`"i=device/router;is=l"`), but not both at the same time (we recommend using the `is` tag.)

8.4.10 Layouting

OMNEST implements an automatic layouting feature, using a variation of the Spring Embedder algorithm. Modules which have not been assigned explicit positions via the `"p="` tag will be automatically placed by the algorithm.

Spring Embedder is a graph layouting algorithm based on a physical model. Graph nodes (modules) repel each other like electric charges of the same sign, and connections act as springs that pull nodes together. There is also friction built in, in order to prevent oscillation of the nodes. The layouting algorithm simulates this physical system until it reaches equilibrium (or times out). The physical rules above have been slightly tweaked to achieve better results.

The algorithm doesn't move any module which has fixed coordinates. Modules that are part of a predefined arrangement (row, matrix, ring, etc., defined via the 3rd and further args of the `"p="` tag) will be moved together, to preserve their relative positions.

> **NOTE:** The positions of modules placed by the layouting algorithm are not available from simulation models. Think about it: what positions should OMNEST report if the model is run under Cmdenv, or under Qtenv but the compound module was never opened in the GUI? The absence of explicit coordinates in the NED file conceptually means that the modeler *doesn't care* about the position of that module.

Caveats:

- If the full graph is too big after layouting, it is scaled back so that it fits on the screen, *unless it contains any fixed-position module*. (For obvious reasons: if there is a module with manually specified position, we don't want to move that one). To prevent rescaling, one can specify a sufficiently large bounding box in the background display string, e.g. `"b=2000,3000"`.
- Submodule size is ignored by the present layouter, so modules with elongated shapes may not be placed ideally.
- The algorithm may produce erratic results, especially for small graphs when the number of submodules is small, or when using predefined (matrix, row, ring, etc) layouts. The *Relayout* toolbar button can then be very useful. Larger networks usually produce satisfactory results.
- The algorithm starts by placing the nodes randomly, and this initial arrangement greatly affects the end result. The algorithm has its own RNG that starts from a default seed. The *Relayout* button changes this seed, and this seed is persistently stored so later runs of the model will produce the same layout.

8.4.11 Changing Display Strings at Runtime

It is often useful to manipulate the display string at runtime. Changing colors, icon, or text may convey status change, and changing a module's position is useful when simulating mobile networks.

Display strings are stored in `cDisplayString` objects inside channels, modules and gates. `cDisplayString` also lets one manipulate the string.

As far as `cDisplayString` is concerned, a display string (e.g. `"p=100,125;i=cloud"`) is a string that consist of several *tags* separated by semicolons, and each tag has a *name* and after an equal sign, zero or more *arguments* separated by commas.

The class facilitates tasks such as finding out what tags a display string has, adding new tags, adding arguments to existing tags, removing tags or replacing arguments. The internal storage method allows very fast operation; it will generally be faster than direct string manipulation. The class doesn't try to interpret the display string in any way, nor does it know the meaning of the different tags; it merely parses the string as data elements separated by semicolons, equal signs and commas.

To get a pointer to a `cDisplayString` object, one can call the components's `getDisplayString()` method.

> **NOTE:** The connection display string is stored in the channel object, but it can also be accessed via the source gate of the connection.

The display string can be overwritten using the `parse()` method. Tag arguments can be set with `setTagArg()`, and tags removed with `removeTag()`.

The following example sets a module's position, icon and status icon in one step:

```
cDisplayString& dispStr = getDisplayString();
dispStr.parse("p=40,20;i=device/cellphone;i2=status/disconnect");
```

Setting an outgoing connection's color to red:

```
cDisplayString& connDispStr = gate("out")->getDisplayString();
connDispStr.parse("ls=red");
```

Setting module background and grid with background display string tags:

```
cDisplayString& parentDispStr = getParentModule()->getDisplayString();
parentDispStr.parse("bgi=maps/europe;bgg=100,2");
```

The following example updates a display string so that it contains the `p=40,20` and `i=device/cellphone` tags:

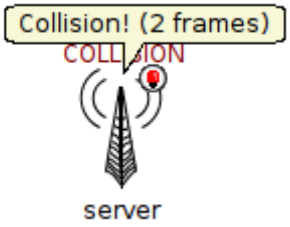
```
dispStr.setTagArg("p", 0, 40);
dispStr.setTagArg("p", 1, 20);
dispStr.setTagArg("i", 0, "device/cellphone");
```

8.5 Bubbles

Modules can display a transient bubble with a short message (e.g. "Going down" or "Connection estalished") by calling the `bubble()` method of `cComponent`. The method takes the string to be displayed as a `const char *` pointer.

An example:

```
bubble("Going down!");
```



If the module often displays bubbles, it is recommended to make the corresponding code conditional on `hasGUI()`. The `hasGUI()` method returns *false* if the simulation is running under Cmdenv.

```
if (hasGUI()) {
    char text[32];
    sprintf(text, "Collision! (%s frames)", numCollidingFrames);
    bubble(text);
}
```

8.6 The Canvas

8.6.1 Overview

The canvas is the 2D drawing API of OMNEST. Using the canvas, one can display lines, curves, polygons, images, text items and their combinations, using colors, transparency, geometric transformations, antialiasing and more. Drawings created with the canvas API can be viewed when the simulation is run under a graphical user interface like Qtenv.

Use cases for the canvas API include displaying textual annotations, status information, live statistics in the form of plots, charts, gauges, counters, etc. Other types of simulations may call for different types of graphical presentation. For example, in mobile and wireless simulations, the canvas API can be used to draw the scene including a background (like a street map or floor plan), mobile objects (vehicles, people), obstacles (trees, buildings, hills), antennas with orientation, and also extra information like connectivity graph, movement trails, individual transmissions and so on.

An arbitrary number of drawings (canvases) can be created, and every module already has one by default. A module's default canvas is the one on which the module's submodules and internal connections are also displayed, so the canvas API can be used to enrich the default, display string based presentation of a compound module.

OMNEST calls the items that appear on a canvas *figures*. The corresponding C++ types are `cCanvas` and `cFigure`. In fact, `cFigure` is an abstract base class, and different kinds of figures are represented by various subclasses of `cFigure`.

Figures can be declared statically in NED files using `@figure` properties, and can also be accessed, created and manipulated programmatically at runtime.

8.6.2 Creating, Accessing and Viewing Canvases

A canvas is represented by the `cCanvas` C++ class. A module's default canvas can be accessed with the `getCanvas()` method of `cModule`. For example, a toplevel submodule can get hold of the network's canvas with the following line:

```
cCanvas *canvas = getParentModule()->getCanvas();
```

Using the canvas pointer, it is possible to check what figures it contains, add new figures, manipulate existing ones, and so on.

New canvases can be created by simply creating new `cCanvas` objects, like so:

```
cCanvas *canvas = new cCanvas("liveStatistics"); // arbitrary name string
```

To view the contents of these additional canvases in Qtenv, one needs to navigate to the canvas' owner object (which will usually be the module that created the canvas), view the list of objects it contains, and double-click the canvas in the list. Giving meaningful names to extra canvas objects like in the example above can simplify the process of locating them in the Qtenv GUI.

8.6.3 Figure Classes

The base class of all figure classes is `cFigure`. The class hierarchy is shown in figure 8.4.

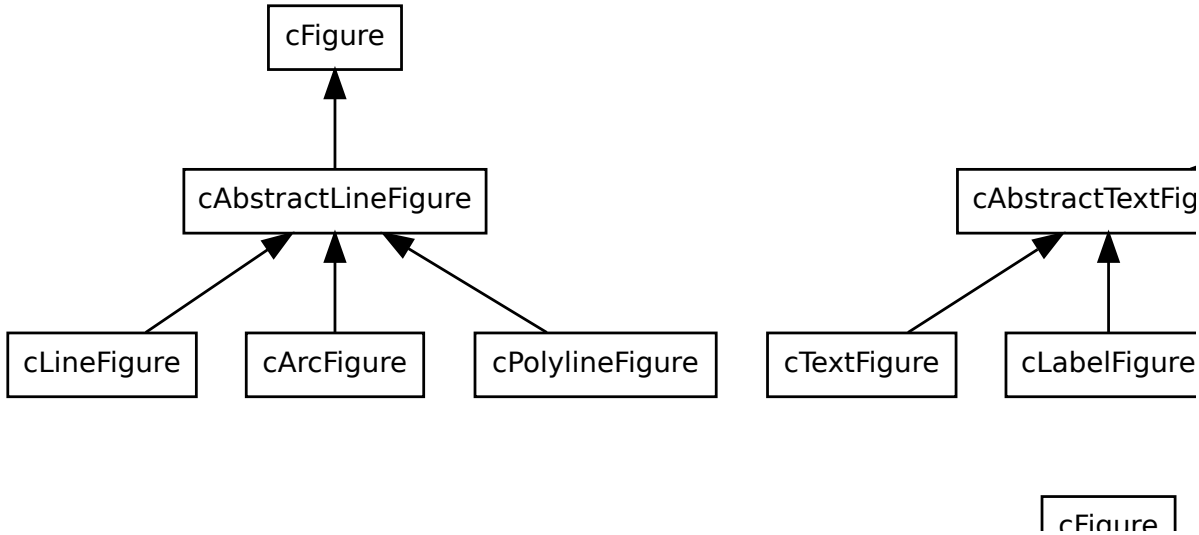


Figure 8.4: cFigure class hierarchy

In subsequent sections, we'll first describe features that are common to all figures, then we'll briefly cover each figure class. Finally, we'll look into how one can define new figure types.

NOTE: Figures are only data storage classes. The real drawing code is inside Qtenv; it might involve a parallel data structure, figure renderer classes, etc. When the canvas is not viewed, corresponding objects in Qtenv do not exist. Therefore, data flow is largely one-directional – figures-to-GUI.

8.6.4 The Figure Tree

Figures on a canvas are organized into a tree. The canvas has a (hidden) *root figure*, and all toplevel figures are children of the root figure. Any figure may contain child figures, not only dedicated ones like `cGroupFigure`.

Every figure also has a name string, inherited from `cNamedObject`. Since figures are in a tree, every figure also has a *hierarchical name*. It consists of the names of figures in the path from the root figure down to the the figure, joined with dots. (The name of the root figure itself is omitted.)

Child figures can be added to a figure with the `addFigure()` method, or inserted into the child list of a figure relative to a sibling with the `insertBefore()` / `insertAfter()` methods. `addFigure()` has two flavours: one for appending, and one for inserting at a numeric position. Child figures can be accessed by name (`getFigure(name)`), or enumerated by index in the child list (`getFigure(k)`, `getNumFigures()`). One can obtain the index of a child figure using `findFigure()`. The `removeFromParent()` method can be used to remove a figure from its parent.

For convenience, `cCanvas` also has `addFigure()`, `getFigure()`, `getNumFigures()` and other methods for managing toplevel figures without the need to go via the root figure.

The following code enumerates the children of a figure named `"group1"`:

```
cFigure *parent = canvas->getFigure("group1");
ASSERT(parent != nullptr);
for (int i = 0; i < parent->getNumFigures(); i++)
    EV << parent->getFigure(i)->getName() << endl;
```

It is also possible to locate a figure by its hierarchical name (`getFigureByPath()`), and to find figure by its (non-hierarchical) name anywhere in a figure subtree (`findFigureRecursively()`).

The `dup()` method of figure classes only duplicates the very figure on which it was called. (The duplicate will not have ay children.) To clone a figure including children, use the `dupTree()` method.

8.6.5 Creating and Manipulating Figures from NED and C++

As mentioned earlier, figures can be defined in the NED file, so they don't always need to be created programmatically. This possibility is useful for creating static backgrounds or statically defining placeholders for dinamically displayed items, among others. Figures defined from NED can be accessed and manipulated from C++ code in the same way as dynamically created ones.

Figures are defined in NED by adding `@figure` properties to a module definition. The hierarchical name of the figure goes into the property index, that is, in square brackets right after `@figure`. The parent of the figure must already exist, that is, when defining `foo.bar.baz`, both `foo` and `foo.bar` must have already been defined (in NED).

Type and various attributes of the figure go into property body, as *key-valuelist* pairs. `type=line` creates a `cLineFigure`, `type=rectangle` creates a `cRectangleFigure`, `type=text` creates a `cTextFigure`, and so on; the list of accepted types is given in appendix H. Further attributes largely correspond to getters and setters of the C++ class denoted by the `type` attribute.

The following example creates a green rectangle and the text *"placeholder"* in it in NED, and the subsequent C++ code changes the same text to *"Hello World!"*.

NED part:

```
module Foo
{
    @display("bgb=800,500");
    @figure[box](type=rectangle; coords=10,50; size=200,100; fillColor=green);
    @figure[box.label](type=text; coords=20,80; text=placeholder);
}
```

And the C++ part:

```
// we assume this code runs in a submodule of the above "Foo" module
cCanvas *canvas = getParentModule()->getCanvas();

// obtain the figure pointer by hierarchical name, and change the text:
cFigure *figure = canvas->getFigureByPath("box.label")
cTextFigure *textFigure = check_and_cast<cTextFigure *>(figure);
textFigure->setText("Hello World!");
```

8.6.6 Stacking Order

The stacking order (a.k.a. Z-order) of figures is jointly determined by the child order and the `cFigure` attribute called Z-index, with the latter taking priority. Z-index is not used directly, but an *effective Z-index* is computed instead, as the *sum* of the Z-index values of the figure and all its ancestors up to the root figure.

A figure with a larger effective Z-index will be displayed above figures with smaller effective Z-indices, regardless of their positions in the figure tree. Among figures whose effective Z-indices are equal, child order determines the stacking order. If two such figures are siblings, the one that occurs *later* in the child list will be drawn above the other. For figures that are not siblings, the child order within the first common ancestor matters.

There are several methods for managing stacking order: `setZIndex()`, `getZIndex()`, `getEffectiveZIndex()`, `insertAbove()`, `insertBelow()`, `isAbove()`, `isBelow()`, `raiseAbove()`, `lowerBelow()`, `raiseToTop()`, `lowerToBottom()`.

8.6.7 Transforms

One of the most powerful features of the Canvas API is being able to assign geometric transformations to figures. OMNEST uses 2D homogeneous transformation matrices, which are able to express affine transforms such as translation, scaling, rotation and skew (shearing). The transformation matrix used by OMNEST has the following format:

$$T = \begin{pmatrix} a & c & t_1 \\ b & d & t_2 \\ 0 & 0 & 1 \end{pmatrix}$$

In a nutshell, given a point with its (x, y) coodinates, one can obtain the transformed version of it by multiplying the transformation matrix by the $(x\ y\ 1)$ column vector (a.k.a. homogeneous coordinates), and dropping the third component:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & t_1 \\ b & d & t_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

The result is the point $(ax + cy + t_1, bx + dy + t_2)$. As one can deduce, a, b, c, d are responsible for rotation, scaling and skew, and t_1 and t_2 for translation. Also, transforming a point by matrix T_1 and then by T_2 is equivalent to transforming the point by the matrix T_2T_1 due to the associativity of matrix multiplication.

The Transform Class

Transformation matrices are represented in OMNEST by the `cFigure::Transform` class.

A `cFigure::Transform` transformation matrix can be initialized in several ways. First, it is possible to assign its `a`, `b`, `c`, `d`, `t1`, `t2` members directly (they are public), or via a six-argument constructor. However, it is usually more convenient to start from the identity transform (as created by the default constructor), and invoke one or more of its several `scale()`, `rotate()`, `skewx()`, `skewy()` and `translate()` member functions. They update the matrix to (also) perform the given operation (scaling, rotation, skewing or translation), as if left-multiplied by a temporary matrix that corresponds to the operation.

The `multiply()` method allows one to combine transformations: `t1.multiply(t2)` sets `t1` to the product `t2*t1`.

To transform a point (represented by the class `cFigure::Point`), one can use the `applyTo()` method of `Transform`. The following code fragment should clarify this:

```
// allow Transform and Point to be referenced without the cFigure:: prefix
typedef cFigure::Transform Transform;
typedef cFigure::Point Point;

// create a matrix that scales by 2, rotates by 45 degrees, and translates by (100
Transform t = Transform().scale(2.0).rotate(M_PI/4).translate(100,0);

// apply the transform to the point (10, 20)
Point p(10, 20);
Point p2 = t.applyTo(p);
```

Figure Transforms

Every figure has an associated transformation matrix, which affects how the figure and its figure subtree are displayed. In other words, the way a figure displayed is affected by its own transformation matrix and the transformation matrices of all of its ancestors, up to the root figure of the canvas. The effective transform will be the product of those transformation matrices.

A figure's transformation matrix is directly accessible via `cFigure`'s `getTransform()`, `setTransform()` member functions. For convenience, `cFigure` also has several `scale()`, `rotate()`, `skewx()`, `skewy()` and `translate()` member functions, which directly operate on the internal transformation matrix.

Some figures have visual aspects that are not, or only optionally affected by the transform. For example, the size and orientation of the text displayed by `cLabelFigure`, in contrast to that of `cTextFigure`, is unaffected by transforms (and of manual zoom as well). Only the position is transformed.

Transform vs move()

In addition to the `translate()`, `scale()`, `rotate()`, etc. functions that update the figure's transformation matrix, figures also have a `move()` method. `move()`, like `translate()`, also moves the figure by a dx, dy offset. However, `move()` works by changing the figure's coordinates, and not by changing the transformation matrix.

Since every figure class stores and interprets its position differently, `move()` is defined for each figure class independently. For example, `cPolylineFigure`'s `move()` changes the coordinates of each point.

`move()` is recursive, that is, it not only moves the figure on which it was called, but also its children. There is also a non-recursive variant, called `moveLocal()`.

8.6.8 Showing/Hiding Figures

Visibility Flag

Figures have a visibility flag that controls whether the figure is displayed. Hiding a figure via the flag will hide the whole figure subtree, not just the figure itself. The flag can be accessed via the `isVisible()`, `setVisible()` member functions of `cFigure`.

Tags

Figures can also be assigned a number of textual tags. Tags do not directly affect rendering, but graphical user interfaces that display canvas content, like Qtenv, offer functionality to interactively show/hide figures based on tags they contain. This GUI figure filter allows one to express conditions like *"Show only figures that have tag `foo` or `bar`, but among them, hide those that also contain tag `baz`"*. Tag-based filtering and the visibility flag are in AND relationship – figures hidden via `setVisible(false)` cannot be displayed using tags. Also when a figure is hidden using the tag filter, its figure subtree will also be hidden.

The tag list of a figure can be accessed with the `getTags()` and `setTags()` `cFigure` methods. They return/accept a single string that contains the tags separated by spaces (a tag itself cannot contain a space.)

Tags functionality, when used carefully, allows one to define "layers" that can be turned on/off from Qtenv.

8.6.9 Figure Tooltip, Associated Object

Tooltip

Figures may be assigned a tooltip text using the `setTooltip()` method. The tooltip is shown in the runtime GUI when one hovers with the mouse over the figure.

Associated Object

In the visualization of many simulations, some figures correspond to certain objects in the simulation model. For example, a truck image may correspond to a module that represents the mobile node in the simulation. Or, an inflating disc that represents a wireless signal may correspond to a message (`cMessage`) in the simulation.

One can set the associated object on a figure using the `setAssociatedObject()` method. The GUI can use this information provide a shortcut access to the associated object, for example select the object in an inspector when the user clicks the figure, or display the object's tooltip over the figure if it does not have its own.

> **CAUTION:** The object must exist (i.e. must not be deleted) while it is associated with the figure. When the object is deleted, the user is responsible for letting the figure forget the pointer, e.g. by a `setAssociatedObject(nullptr)` call.

8.6.10 Specifying Positions, Colors, Fonts and Other Properties

Points

Points are represented by the `cFigure::Point` struct:

```
struct Point {
    double x, y;
    ...
};
```

In addition to the public `x`, `y` members and a two-argument constructor for convenient initialization, the struct provides overloaded operators (+,-,*,/) and some utility functions like `translate()`, `distanceTo()` and `str()`.

Rectangles

Rectangles are represented by the `cFigure::Rectangle` struct:

```
struct Rectangle {
    double x, y,
    double width, height;
    ...
};
```

A rectangle is specified with the coordinates of their top-left corner, their width and height. The latter two are expected to be nonnegative. In addition to the public `x`, `y`, `width`, `height` members and a four-argument constructor for convenient initialization, the struct also has utility functions like `getCenter()`, `getSize()`, `translate()` and `str()`.

Colors

Colors are represented by the `cFigure::Color` struct as 24-bit RGB colors:

```
struct Color {
    uint8_t red, green, blue;
```

```
    ...
};
```

In addition to the public `red`, `green`, `blue` members and a three-argument constructor for convenient initialization, the struct also has a string-based constructor and `str()` function. The string form accepts various notations: HTML colors (`#rrggbb`), HSB colors in a similar notation (`@hhssbb`), and English color names (SVG and X11 color names, to be more precise.)

However, one doesn't need to use `Color` directly. There are also predefined constants for the basic colors (`BLACK`, `WHITE`, `GREY`, `RED`, `GREEN`, `BLUE`, `YELLOW`, `CYAN`, `MAGENTA`), as well as a collection of carefully chosen dark and light colors, suitable for e.g. chart drawing, in the arrays `GOOD_DARK_COLORS[]` and `GOOD_LIGHT_COLORS[]`; for convenience, the number of colors in each are in the `NUM_GOOD_DARK_COLORS` and `NUM_GOOD_LIGHT_COLORS` constants).

The following ways of specifying colors are all valid:

```
cFigure::BLACK;
cFigure::Color("steelblue");
cFigure::Color("#3d7a8f");
cFigure::Color("@20ff80");
cFigure::GOOD_DARK_COLORS[2];
cFigure::GOOD_LIGHT_COLORS[intrand(NUM_GOOD_LIGHT_COLORS)];
```

Fonts

The requested font for text figures is represented by the `cFigure::Font` struct. It stores the typeface, font style and font size in one.

```
struct Font {
    std::string typeface;
    int pointSize;
    uint8_t style;
    ...
};
```

The font does not need to be fully specified, there are some defaults. When `typeface` is set to the empty string or when `pointSize` is zero or a negative value, that means that the default font or the default size should be used, respectively.

The `style` field can be either `FONT_NONE`, or the binary OR of the following constants: `FONT_BOLD`, `FONT_ITALIC`, `FONT_UNDERLINE`.

The struct also has a three-argument constructor for convenient initialization, and an `str()` function that returns a human-readable text representation of the contents.

Some examples:

```
cFigure::Font("Arial");  // default size, normal
cFigure::Font("Arial", 12);  // 12pt, normal
cFigure::Font("Arial", 12, cFigure::FONT_BOLD | cFigure::FONT_ITALIC);
```

Other Line and Shape Properties

`cFigure` also contains a number of enums as inner types to describe various line, shape, text and image properties. Here they are:

LineStyle

Values: `LINE_SOLID`, `LINE_DOTTED`, `LINE_DASHED`

This enum (`cFigure::LineStyle`) is used by line and shape figures to determine their line/border style. The precise graphical interpretation, e.g. dash lengths for the *dashed* style, depends on the graphics library that the GUI was implemented with.

CapStyle

Values: `CAP_BUTT`, `CAP_ROUND`, `CAP_SQUARE`

This enum is used by line and path figures, and it indicates the shape to be used at the end of the lines or open subpaths.

JoinStyle

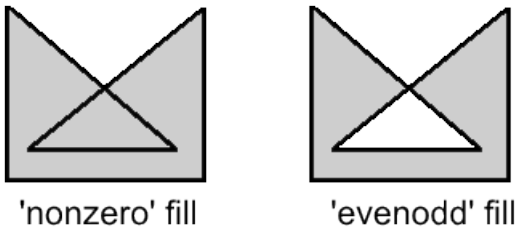
Values: `JOIN_BEVEL`, `JOIN_ROUND`, `JOIN_MITER`

This enum indicates the shape to be used when two line segments are joined, in line or shape figures.

FillRule

Values: `FILL_EVENODD`, `FILL_NONZERO`.

This enum determines which regions of a self-intersecting shape should be considered to be inside the shape, and thus be filled.



Arrowhead

Values: `ARROW_NONE`, `ARROW_SIMPLE`, `ARROW_TRIANGLE`, `ARROW_BARBED`.

Some figures support displaying arrowheads at one or both ends of a line. This enum determines the style of the arrowhead to be used.

Interpolation

Values: `INTERPOLATION_NONE`, `INTERPOLATION_FAST`, `INTERPOLATION_BEST`.

Interpolation is used for rendering an image when it is not displayed at its native resolution. This enum indicates the algorithm to be used for interpolation.

The mode *none* selects the "nearest neighbor" algorithm. *Fast* emphasizes speed, and *best* emphasizes quality; however, the exact choice of algorithm (bilinear, bicubic, quadratic, etc.) depends on features of the graphics library that the GUI was implemented with.

Anchor

Values: `ANCHOR_CENTER`, `ANCHOR_N`, `ANCHOR_E`, `ANCHOR_S`, `ANCHOR_W`, `ANCHOR_NW`, `ANCHOR_NE`, `ANCHOR_SE`, `ANCHOR_SW`; `ANCHOR_BASELINE_START`, `ANCHOR_BASELINE_MIDDLE`, `ANCHOR_BASELINE_END`.

Some figures like text and image figures are placed by specifying a single point (*position*) plus an anchor mode, a value from this enum. The anchor mode indicates which point of the bounding box of the figure should be positioned over the specified point. For example, when using `ANCHOR_N`, the figure is placed so that its top-middle point falls at the specified point.

The last three, *baseline* constants are only used with text figures, and indicate that the start, middle or end of the text's baseline is the anchor point.

8.6.11 Primitive Figures

Now that we know all about figures in general, we can look into the specific figure classes provided by OMNEST.

cAbstractLineFigure

`cAbstractLineFigure` is the common base class for various line figures, providing line color, style, width, opacity, arrowhead and other properties for them.

Line color can be set with `setLineColor()`, and line width with `setLineWidth()`. Lines can be solid, dashed, dotted, etc.; line style can be set with `setLineStyle()`. The default line color is black.

Lines can be partially transparent. This property can be controlled with `setLineOpacity()` that takes a `double` between 0 and 1: a zero argument means fully transparent, and one means fully opaque.

Lines can have various cap styles: butt, square, round, etc., which can be selected with `setCapStyle()`. Join style, which is a related property, is not part of `cAbstractLineFigure` but instead added to specific subclasses where it makes sense.

Lines may also be augmented with arrowheads at either or both ends. Arrowheads can be selected with `setStartArrowhead()` and `setEndArrowhead()`.

Transformations such as scaling or skew do affect the width of the line as it is rendered on the canvas. Whether zooming (by the user) should also affect it can be controlled by setting a flag (`setZoomLineWidth()`). The default is non-zooming lines.

Specifying zero for line width is currently not allowed. To hide the line, use `setVisible(false)`.[4]

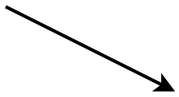
cLineFigure

`cLineFigure` displays a single straight line segment. The endpoints of the line can be set with the `setStart()`/`setEnd()` methods. Other properties such as color and line style are inherited from `cAbstractLineFigure`.

An example that draws an arrow from (0,0) to (100,100):

```
cLineFigure *line = new cLineFigure("line");
line->setStart(cFigure::Point(0,0));
line->setEnd(cFigure::Point(100,50));
line->setLineWidth(2);
line->setEndArrowhead(cFigure::ARROW_BARBED);
```

The result:



cArcFigure

`cArcFigure` displays an axis-aligned arc. (To display a non-axis-aligned arc, apply a transform to `cArcFigure`, or use `cPathFigure`.) The arc's geometry is determined by the bounding box of the circle or ellipse, and a start and end angle; they can be set with the `setBounds()`, `setStartAngle()` and `setEndAngle()` methods. Other properties such as color and line style are inherited from `cAbstractLineFigure`.

For angles, zero points east. Angles that go counterclockwise are positive, and those that go clockwise are negative.

> **NOTE:** Angles are in radians in the C++ API, but in degrees when the figure is defined in the NED file via `@figure`.

Here is an example that draws a blue arc with an arrowhead that goes counter-clockwise from 3 hours to 12 hours on the clock:

```
cArcFigure *arc = new cArcFigure("arc");
arc->setBounds(cFigure::Rectangle(10,10,100,100));
arc->setStartAngle(0);
arc->setEndAngle(M_PI/2);
arc->setLineColor(cFigure::BLUE);
arc->setEndArrowhead(cFigure::ARROW_BARBED);
```

The result:

[4] It would make sense to display zero-width lines as hairlines that are always rendered as one pixel wide regardless of transforms and zoom level, but that is not possible on all platforms.

cPolylineFigure

By default, `cPolylineFigure` displays multiple connecting straight line segments. The class stores geometry information as a sequence of points. The line may be *smoothed*, so the figure can also display complex curves.

The points can be set with `setPoints()` that takes `std::vector<Point>`, or added one-by-one using `addPoint()`. Elements in the point list can be read and overwritten (`getPoint()`, `setPoint()`). One can also insert and remove points (`insertPoint()` and `removePoint()`.

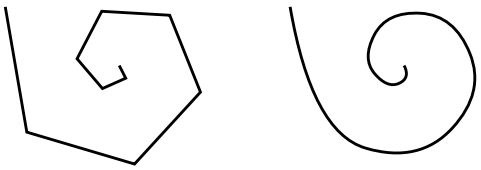
A smoothed line is drawn as a series of Bezier curves, which touch the start point of the first line segment, the end point of the last line segment, and the midpoints of intermediate line segments, while intermediate points serve as control points. Smoothing can be turned on/off using `setSmooth()`.

Additional properties such as color and line style are inherited from `cAbstractLineFigure`. Line join style (which is not part of `cAbstractLineFigure`) can be set with `setJoinStyle()`.

Here is an example that uses a smoothed polyline to draw a spiral:

```
cPolylineFigure *polyline = new cPolylineFigure("polyline");
const double C = 1.1;
for (int i = 0; i < 10; i++)
    polyline->addPoint(cFigure::Point(5*i*cos(C*i), 5*i*sin(C*i)));
polyline->move(100, 100);
polyline->setSmooth(true);
```

The result, with both *smooth=false* and *smooth=true*:



cAbstractShapeFigure

`cAbstractShapeFigure` is an abstract base class for various shapes, providing line and fill color, line and fill opacity, line style, line width, and other properties for them.

Both outline and fill are optional, they can be turned on and off independently with the `setOutlined()` and `setFilled()` methods. The default is outlined but unfilled shapes.

Similar to `cAbstractLineFigure`, line color can be set with `setLineColor()`, and line width with `setLineWidth()`. Lines can be solid, dashed, dotted, etc.; line style can be set with `setLineStyle()`. The default line color is black.

Fill color can be set with `setFillColor()`. The default fill color is blue (although it is indifferent until one calls `setFilled(true)`).

> **NOTE:** Invoking `setFillColor()` alone does not make the shape filled, one also needs to call `setFilled(true)` for that.

Shapes can be partially transparent, and opacity can be set individually for the outline and the fill, using `setLineOpacity()` and `setFillOpacity()`. These functions accept a `double` between 0 and 1: a zero argument means fully transparent, and one means fully opaque.

When the outline is drawn with a width larger than one pixel, it will be drawn symmetrically, i.e. approximately 50-50% of its width will fall inside and outside the shape. (This also means that the fill and a wide outline will partially overlap, but that is only apparent if the outline is also partially transparent.)

Transformations such as scaling or skew do affect the width of the line as it is rendered on the canvas. Whether zooming (by the user) should also affect it can be controlled by setting a flag (`setZoomLineWidth()`). The default is non-zooming lines.

Specifying zero for line width is currently not allowed. To hide the outline, `setOutlined(false)` can be used.

cRectangleFigure

`cRectangleFigure` displays an axis-aligned rectangle with optionally rounded corners. As with all shape figures, drawing of both the outline and the fill are optional. Line and fill color, and several other properties are inherited from `cAbstractShapeFigure`.

The figure's geometry can be set with the `setBounds()` method that takes a `cFigure::Rectangle`. The radii for the rounded corners can be set independently for the x and y direction using `setCornerRx()` and `setCornerRy()`, or together with `setCornerRadius()`.

The following example draws a rounded rectangle of size 160x100, filled with a "good dark color".

```
cRectangleFigure *rect = new cRectangleFigure("rect");
rect->setBounds(cFigure::Rectangle(100,100,160,100));
rect->setCornerRadius(5);
rect->setFilled(true);
rect->setFillColor(cFigure::GOOD_LIGHT_COLORS[0]);
```

The result:

cOvalFigure

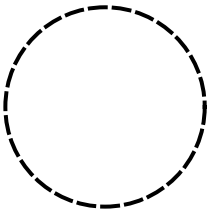
`cOvalFigure` displays a circle or an axis-aligned ellipse. As with all shape figures, drawing of both the outline and the fill are optional. Line and fill color, and several other properties are inherited from `cAbstractShapeFigure`.

The geometry is specified with the bounding box, and it can be set with the `setBounds()` method that takes a `cFigure::Rectangle`.

The following example draws a circle of diameter 120 with a wide dotted line.

```
cOvalFigure *circle = new cOvalFigure("circle");
circle->setBounds(cFigure::Rectangle(100,100,120,120));
circle->setLineWidth(2);
circle->setLineStyle(cFigure::LINE_DOTTED);
```

The result:



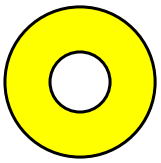
cRingFigure

`cRingFigure` displays a ring, with explicitly controllable inner/outer radii. The inner and outer circles (or ellipses) form the outline, and the area between them is filled. As with all shape figures, drawing of both the outline and the fill are optional. Line and fill color, and several other properties are inherited from `cAbstractShapeFigure`.

The geometry is determined by the bounding box that defines the outer circle, and the x and y radii of the inner oval. They can be set with the `setBounds()`, `setInnerRx()` and `setInnerRy()` member functions. There is also a utility method for setting both inner radii together, named `setInnerRadius()`.

The following example draws a ring with an outer diameter of 50 and inner diameter of 20.

```
cRingFigure *ring = new cRingFigure("ring");
ring->setBounds(cFigure::Rectangle(100,100,50,50));
ring->setInnerRadius(10);
ring->setFilled(true);
ring->setFillColor(cFigure::YELLOW);
```



cPieSliceFigure

`cPieSliceFigure` displays a pie slice, that is, a section of an axis-aligned disc or filled ellipse. The outline of the pie slice consists of an arc and two radii. As with all shape figures, drawing of both the outline and the fill are optional.

Similar to an arc, a pie slice is determined by the bounding box of the full disc or ellipse, and a start and an end angle. They can be set with the `setBounds()`, `setStartAngle()` and `setEndAngle()` methods.

For angles, zero points east. Angles that go counterclockwise are positive, and those that go clockwise are negative.

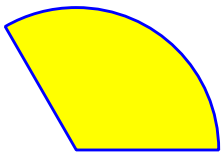
> **NOTE:** Angles are in radians in the C++ API, but in degrees when the figure is defined in the NED file via `@figure`.

Line and fill color, and several other properties are inherited from `cAbstractShapeFigure`.

The following example draws pie slice that's one third of a whole pie:

```
cPieSliceFigure *pieslice = new cPieSliceFigure("pieslice");
pieslice->setBounds(cFigure::Rectangle(100,100,100,100));
pieslice->setStartAngle(0);
pieslice->setEndAngle(2*M_PI/3);
pieslice->setFilled(true);
pieslice->setLineColor(cFigure::BLUE);
pieslice->setFillColor(cFigure::YELLOW);
```

The result:



cPolygonFigure

`cPolygonFigure` displays a (closed) polygon, determined by a sequence of points. The polygon may be *smoothed*. A smoothed polygon is drawn as a series of cubic Bezier curves, where the curves touch the midpoints of the sides, and vertices serve as control points. Smoothing can be turned on/off using `setSmooth()`.

The points can be set with `setPoints()` that takes `std::vector<Point>`, or added one-by-one using `addPoint()`. Elements in the point list can be read and overwritten (`getPoint()`, `setPoint()`). One can also insert and remove points (`insertPoint()` and `removePoint()`.

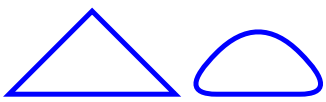
As with all shape figures, drawing of both the outline and the fill are optional. The drawing of filled self-intersecting polygons is controlled by the fill rule, which defaults to even-odd (`FILL_EVENODD`), and can be set with `setFillRule()`. Line join style can be set with the `setJoinStyle()`.

Line and fill color, and several other properties are inherited from `cAbstractShapeFigure`.

Here is an example of a smoothed polygon that also demonstrates the use of `setPoints()`:

```
cPolygonFigure *polygon = new cPolygonFigure("polygon");
std::vector<cFigure::Point> points;
points.push_back(cFigure::Point(0, 100));
points.push_back(cFigure::Point(50, 100));
points.push_back(cFigure::Point(100, 100));
points.push_back(cFigure::Point(50, 50));
polygon->setPoints(points);
polygon->setLineColor(cFigure::BLUE);
polygon->setLineWidth(3);
polygon->setSmooth(true);
```

The result, with both *smooth=false* and *smooth=true*:



cPathFigure

`cPathFigure` displays a "path", a complex shape or line modeled after SVG paths. A path may consist of any number of straight line segments, Bezier curves and arcs. The path can be disjoint as well. Closed paths may be filled. The drawing of filled self-intersecting polygons is controlled by the *fill rule* property. Line and fill color, and several other properties are inherited from `cAbstractShapeFigure`.

A path, when given as a string, looks like this one that draws a triangle:

M 150 0 L 75 200 L 225 200 Z

It consists of a sequence of commands (*M* for *moveto*, *L* for *lineto*, etc.) that are each followed by numeric parameters (except *Z*). All commands can be expressed with lowercase letter, too. A capital letter means that the target point is given with *absolute* coordinates, a lowercase letter means they are given *relative* to the target point of the previous command.

`cPathFigure` can accept the path in string form (`setPath()`), or one can assemble the path with a series of method calls like `addMoveTo()`. The path can be cleared with the `clearPath()` method.

The commands with argument list and the corresponding *add* methods:

- **M** *x y*: move; `addMoveTo()`, `addMoveRel()`
- **L** *x y*: line; `addLineTo()`, `addLineRel()`
- **H** *x*: horizontal line; `addHorizontalLineTo()`, `addHorizontalLineRel()`
- **V** *y*: vertical line; `addVerticalLineTo()`, `addVerticalLineRel()`
- **A** *rx ry phi largeArc sweep x y*: arc; `addArcTo()`, `addArcRel()`
- **Q** *x1 y1 x y*: curve; `addCurveTo()`, `addCurveRel()`
- **T** *x y*: smooth curve; `addSmoothCurveTo()`, `addSmoothCurveRel()`
- **C** *x1 y1 x2 y2 x y*: cubic Bezier curve; `addCubicBezierCurveTo()`, `addCubicBezier-CurveRel()`
- **S** *x1 y1 x y*: smooth cubic Bezier curve; `addSmoothCubicBezierCurveTo()`, `addSmoothCu-bicBezierCurveRel()`
- **Z**: close path; `addClosePath()`

In the parameter lists, (x, y) are the target points (substitute (dx, dy) for the lowercase, relative versions.) For the Bezier curves, $x1, y1$ and $(x2, y2)$ are control points. For the arc, rx and ry are the radii of the ellipse, phi is a rotation angle in degrees for the ellipse, and $largeArc$ and $sweep$ are both booleans (0 or 1) that select which portion of the ellipse should be taken.[5]

No matter how the path was created, the string form can be obtained with the `getPath()` method, and the parsed form with the `getNumPathItems()`, `getPathItem(k)` methods. The latter returns pointer to a `cPathFigure::PathItem`, which is a base class with subclasses for every item type.

Line join style, cap style (for open subpaths), and fill rule (for closed subpaths) can be set with the `setJoinStyle()`, `setCapStyle()`, `setFillRule()` methods.

[5]For more details, consult the SVG specification.

cPathFigure has one more property, a (dx, dy) offset, which exists to simplify the implementation of the move() method. Offset causes the figure to be translated by the given amount for drawing. For other figure types, move() directly updates the coordinates, so it is effectively a wrapper for setPosition() or setBounds(). For path figures, implementing move() so that it updates every path item would be cumbersome and potentially also confusing for users. Instead, move() updates the offset. Offset can be set with setOffset(),

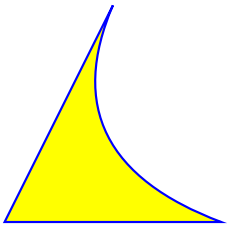
In the first example, the path is given as a string:

```
cPathFigure *path = new cPathFigure("path");
path->setPath("M 0 150 L 50 50 Q 20 120 100 150 Z");
path->setFilled(true);
path->setLineColor(cFigure::BLUE);
path->setFillColor(cFigure::YELLOW);
```

The second example creates the equivalent path programmatically.

```
cPathFigure *path2 = new cPathFigure("path");
path2->addMoveTo(0,150);
path2->addLineTo(50,50);
path2->addCurveTo(20,120,100,150);
path2->addClosePath();
path2->setFilled(true);
path2->setLineColor(cFigure::BLUE);
path2->setFillColor(cFigure::YELLOW);
```

The result:



cAbstractTextFigure

cAbstractTextFigure is an abstract base class for figures that display (potentially multiline) text.

The location of the text on the canvas is determined jointly by a *position* and an *anchor*. The anchor tells how to place the text relative to the positioning point. For example, if anchor is ANCHOR_CENTER then the text is centered on the point; if anchor is ANCHOR_N then the text will be drawn so that its top center point is at the positioning point. The values ANCHOR_BASELINE_START, ANCHOR_BASELINE_MIDDLE, ANCHOR_BASELINE_END refer to the beginning, middle and end of the baseline of the (first line of the) text as anchor point. The member functions to set the positioning point and the anchor are setPosition() and setAnchor(). Anchor defaults to ANCHOR_CENTER.

The font can be set with the setFont() member function that takes cFigure::Font, a class that encapsulates typeface, font style and size. Color can be set with setColor(). The displayed text can also be partially transparent. This is controlled with the setOpacity() member function that accepts an double in the $[0, 1]$ range, 0 meaning fully transparent (invisible), and 1 meaning fully opaque.

It is also possible to have a partially transparent "halo" displayed around the text. The halo improves readability when the text is displayed over a background that has a similar color as the text, or when it overlaps with other text items. The halo can be turned on with `setHalo()`.

cTextFigure

`cTextFigure` displays text which is affected by zooming and transformations. Font, color, position, anchoring and other properties are inherited from `cAbstractTextFigure`.

The following example displays a text in dark blue 12-point bold Arial font.

```
cTextFigure *text = new cTextFigure("text");
text->setText("This is some text.");
text->setPosition(cFigure::Point(100,100));
text->setAnchor(cFigure::ANCHOR_BASELINE_MIDDLE);
text->setColor(cFigure::Color("#000040"));
text->setFont(cFigure::Font("Arial", 12, cFigure::FONT_BOLD));
```

The result:

cLabelFigure

`cLabelFigure` displays text which is unaffected by zooming or transformations, except its position. Font, color, position, anchoring and other properties are inherited from `cAbstractTextFigure`. The angle of the label can be set with the `setAngle()` method. Zero angle means horizontal (unrotated) text. Positive values rotate counterclockwise, while negative values rotate clockwise.

> **NOTE:** Angles are in radians in the C++ API, but in degrees when the figure is defined in the NED file via `@figure`.

The following example displays a label in Courier New with the default size, slightly transparent.

```
cLabelFigure *label = new cLabelFigure("label");
label->setText("This is a label.");
label->setPosition(cFigure::Point(100,100));
label->setAnchor(cFigure::ANCHOR_NW);
label->setFont(cFigure::Font("Courier New"));
label->setOpacity(0.9);
```

The result:

cAbstractImageFigure

`cAbstractImageFigure` is an abstract base class for image figures.

The location of the image on the canvas is determined jointly by a *position* and an *anchor*. The anchor tells how to place the image relative to the positioning point. For example, if anchor is `ANCHOR_CENTER` then the image is centered on the point; if anchor is `ANCHOR_N` then the image will be drawn so that its top center point is at the positioning point. The member functions to set the positioning point and the anchor are `setPosition()` and `setAnchor()`. Anchor defaults to `ANCHOR_CENTER`.

By default, the figure's width/height will be taken from the image's dimensions in pixels. This can be overridden with the`setWidth()` / `setHeight()` methods, causing the image to be scaled. `setWidth(0)` / `setHeight(0)` reset the default (automatic) width and height.

One can choose from several interpolation modes that control how the image is rendered when it is not drawn in its natural size. Interpolation mode can be set with `setInterpolation()`, and defaults to `INTERPOLATION_FAST`.

Images can be tinted; this feature is controlled by a tint color and a tint amount, a $[0,1]$ real number. They can be set with the `setTintColor()` and `setTintAmount()` methods, respectively.

Images may also be rendered as partially transparent, which is controlled by the opacity property, a $[0,1]$ real number. Opacity can be set with the `setOpacity()` method. The rendering process will combine this property with the transparency information contained within the image, i.e. the alpha channel.

cImageFigure

`cImageFigure` displays an image, typically an icon or a background image, loaded from the OMNEST image path. Positioning and other properties are inherited from `cAbstractImageFigure`. Unlike `cIconFigure`, `cImageFigure` fully obeys transforms and zoom.

The following example displays a map:

```
cImageFigure *image = new cImageFigure("map");
image->setPosition(cFigure::Point(0,0));
image->setAnchor(cFigure::ANCHOR_NW);
image->setImageName("maps/europe");
image->setWidth(600);
image->setHeight(500);
```

cIconFigure

`cIconFigure` displays a non-zooming image, loaded from the OMNEST image path. Positioning and other properties are inherited from `cAbstractImageFigure`.

`cIconFigure` is not affected by transforms or zoom, except its position. (It can still be resized, though, via `setWidth()` / `setHeight()`.)

The following example displays an icon similar to the way the `"i=block/sink,gold,30"` display string tag would, and makes it slightly transparent:

```
cIconFigure *icon = new cIconFigure("icon");
icon->setPosition(cFigure::Point(100,100));
icon->setImageName("block/sink");
icon->setTintColor(cFigure::Color("gold"));
icon->setTintAmount(0.6);
icon->setOpacity(0.8);
```

The result:

cPixmapFigure

`cPixmapFigure` displays a user-defined raster image. A pixmap figure may be used to display e.g. a heat map. Support for scaling and various interpolation modes are useful here. Positioning and other properties are inherited from `cAbstractImageFigure`.

A pixmap itself is represented by the `cFigure::Pixmap` class.

`cFigure::Pixmap` stores a rectangular array of 32-bit RGBA pixels, and allows pixels to be manipulated directly. The size ($width \times height$) as well as the default fill can be specified in the constructor. The pixmap can be resized (i.e. pixels added/removed at the right and/or bottom) using `setSize()`, and it can be filled with a color using `fill()`. Pixels can be directly accessed with `pixel(x,y)`.

A pixel is returned as type `cFigure::RGBA`, which is a convenience struct that, in addition to having the four public `uint8_t` fields (`red`, `green`, `blue`, `alpha`), is augmented with several utility methods.

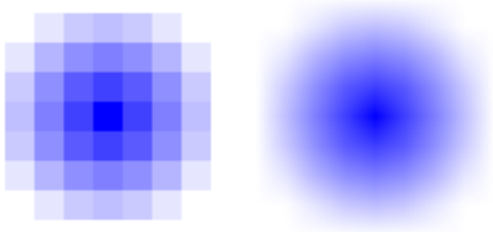
Many `Pixmap` and `RGBA` methods accept or return `cFigure::Color` and opacity, converting between them and `RGBA`. (Opacity is a $[0,1]$ real number that is mapped to the 0..255 alpha channel. 0 means fully transparent, and 1 means fully opaque.)

One can set up and manipulate the image that `cPixmapFigure` displays in two ways. First, one can create and fill a `cFigure::Pixmap` separately, and set it on `cPixmapFigure` using `setPixmap()`. This will overwrite the figure's internal pixmap instance that it displays. The second way is to utilize `cPixmapFigure`'s methods such as `setPixmapSize()`, `fill()`, `setPixel()`, `setPixelColor()`, `setPixelOpacity()`, etc. that delegate to the internal pixmap instance.

The following example displays a small heat map by manipulating the transparency of the pixels. The 9-by-9 pixel image is stretched to 100 units each direction on the screen.

```
cPixmapFigure *pixmapFigure = new cPixmapFigure("pixmap");
pixmapFigure->setPosition(cFigure::Point(100,100));
pixmapFigure->setSize(100, 100);
pixmapFigure->setPixmapSize(9, 9, cFigure::BLUE, 1);
for (int y = 0; y < pixmapFigure->getPixmapHeight(); y++) {
    for (int x = 0; x < pixmapFigure->getPixmapWidth(); x++) {
        double opacity = 1 - sqrt((x-4)*(x-4) + (y-4)*(y-4))/4;
        if (opacity < 0) opacity = 0;
        pixmapFigure->setPixelOpacity(x, y, opacity);
    }
}
pixmapFigure->setInterpolation(cFigure::INTERPOLATION_FAST);
```

The result, both with *interpolation=NONE* and *interpolation=FAST*:



cGroupFigure

`cGroupFigure` is for the sole purpose of grouping its children. It has no visual appearance. The usefulness of a group figure comes from the fact that elements of a group can be hidden / shown together, and also transformations are inherited from parent to child, thus, children of a group can be moved, scaled, rotated, etc. together by updating the group's transformation matrix.

The following example creates a group with two subfigures, then moves and rotates them as one unit.

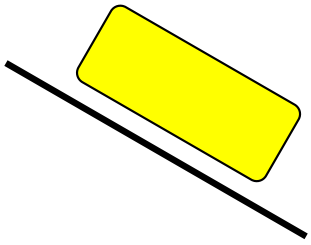
```
cGroupFigure *group = new cGroupFigure("group");

cRectangleFigure *rect = new cRectangleFigure("rect");
rect->setBounds(cFigure::Rectangle(-50,0,100,40));
rect->setCornerRadius(5);
rect->setFilled(true);
rect->setFillColor(cFigure::YELLOW);

cLineFigure *line = new cLineFigure("line");
line->setStart(cFigure::Point(-80,50));
line->setEnd(cFigure::Point(80,50));
line->setLineWidth(3);

group->addFigure(rect);
group->addFigure(line);
group->translate(100, 100);
group->rotate(M_PI/6, 100, 100);
```

The result:



cPanelFigure

`cPanelFigure` is similar to `cGroupFigure` in that it is also intended for grouping its children and has no visual appearance of its own. However, it has a special behavior regarding

transformations and especially zooming.

`cPanelFigure` sets up an axis-aligned, unscaled coordinate system for its children, canceling the effect of any transformation (scaling, rotation, etc.) inherited from ancestor figures. This allows for pixel-based positioning of children, and makes them immune to zooming.

Unlike `cGroupFigure` which itself has position attribute, `cPanelFigure` uses two points for positioning, a *position* and an *anchor point*. Position is interpreted in the coordinate system of the panel figure's parent, while the anchor point is interpreted in the coordinate system of the panel figure itself. To place the panel figure on the canvas, the panel's anchor point is mapped to *position* in the parent.

Setting a transformation on the panel figure itself allows for rotation, scaling, and skewing of its children. The anchor point is also affected by this transformation.

The following example demonstrates `cPanelFigure` behavior. It creates a normal group figure as parent for the panel, and sets up a skewed coordinate system on it. A reference image is also added to it, in order to make the effect of skew visible. The panel figure is also added to it as a child. The panel contains an image (showing the same icon as the reference image), and a border around it.

```
cGroupFigure *layer = new cGroupFigure("parent");
layer->skewx(-0.3);

cImageFigure *referenceImg = new cImageFigure("ref");
referenceImg->setImageName("block/broadcast");
referenceImg->setPosition(cFigure::Point(200,200));
referenceImg->setOpacity(0.3);
layer->addFigure(referenceImg);

cPanelFigure *panel = new cPanelFigure("panel");

cImageFigure *img = new cImageFigure("img");
img->setImageName("block/broadcast");
img->setPosition(cFigure::Point(0,0));
panel->addFigure(img);

cRectangleFigure *border = new cRectangleFigure("border");
border->setBounds(cFigure::Rectangle(-25,-25,50,50));
border->setLineWidth(3);
panel->addFigure(border);

layer->addFigure(panel);
panel->setAnchorPoint(cFigure::Point(0,0));
panel->setPosition(cFigure::Point(210,200));
```

The screenshot shows the result at an approx. 4x zoom level. The large semi-transparent image is the reference image, the smaller one is the image within the panel figure. Note that neither the skew nor the zoom has affected the panel figure's children.

8.6.12 Compound Figures

Any graphics can be built using primitive (i.e. elementary) figures alone. However, when the graphical presentation of a simulation grows complex, it is often convenient to be able to group certain figures and treat them as a single unit. For example, although a bar chart can be displayed using several independent rectangle, line and text items, there are clearly benefits to being able to handle them together as a single bar chart object.

Compound figures are `cFigure` sublasses that are themselves composed of several figures, but can be instantiated and manipulated as a single figure. Compound figure classes can be used from C++ code like normal figures, and can also be made to be able to be instatiated from `@figure` properties.

Compound figure classes usually subclass from `cGroupFigure`. The class would typically maintain pointers to its subfigures in class members, and its methods (getters, setters, etc.) would operate on the subfigures.

To be able to use the new C++ class with `@figure`, it needs to be registered using the `Register_Figure()` macro. The macro expects two arguments: one is the type name by which the figure is known to `@figure` (the string to be used with the `type` property key), and the other is the C++ class name. For example, to be able to instantiate a class named `FooFigure` with `@figure[...](type=foo;...)`, the following line needs to be added into the C++ source:

```
Register_Figure("foo", FooFigure);
```

If the figure needs to be able take values from `@figure` properties, the class needs to override the `parse(cProperty*)` method, and proably also `getAllowedPropertyKeys()`. We recommend that you examine the code of the figure classes built into OMNEST for implementation hints.

8.6.13 Self-Refreshing Figures

Most figures are entirely passive objects. When they need to be moved or updated during the course of the simulation, there must be an active component in the simulation that does it for them. Usually it is the `refreshDisplay()` method of some simple module (or modules) that contain the code that updates various properties of the figures.

However, certain figures can benefit from being able to refresh themselves during the simulation. Picture, for example, a compound figure (see previous section) that displays a line chart

which is continually updated with new data as the simulation progresses. The `LineChartFigure` class may contain an `addDataPoint(x,y)` method which is called from other parts of the simulation to add new data to the chart. The question is when to update the subfigures that make up the chart: the line(s), axis ticks and labels, etc. It is clearly not very efficient to do it in every `addDataPoint(x,y)` call, especially when the simulation is running in Express mode when the screen is not refreshed very often. Luckily, our hypothetical `LineChartFigure` class can do better, and only refresh its subfigures when it matters, i.e. when the result can actually be seen in the GUI. To do that, the class needs to override `cFigure`'s `refreshDisplay()` method, and place the subfigure updating code there.

Figure classes that override `refreshDisplay()` to refresh their own contents are called *self-refreshing figures*. Self-refreshing figures as a feature are available since OMNEST version 5.1.

`refreshDisplay()` is declared on `cFigure` as:

```
virtual void refreshDisplay();
```

The default implementation does nothing.

Like `cModule`'s `refreshDisplay()`, `cFigure`'s `refreshDisplay()` is invoked only under graphical user interfaces (Qtenv), and right before display updates. However, it is only invoked for figures on canvases that are currently displayed. This makes it possible for canvases that are never viewed to have zero refresh overhead.

Since `cFigure`'s `refreshDisplay()` is only invoked when the canvas is visible, it should only be used to update local state, i.e. only local members and local subfigures. The code should certainly not access other canvases, let alone change the state of the simulation.

8.6.14 Figures with Custom Renderers

In rare cases it might be necessary to create figure types where the rendering is entirely custom, and not based on already existing figures. The difficulty arises from the point that figures are only data storage classes, actual drawing takes place in the GUI library such as Qtenv. Thus, in addition to writing the new figure class, one also needs to extend Qtenv with the corresponding rendering code. We won't go into full details on how to extend Qtenv here, just give you a few pointers in case you need it.

In Qtenv, rendering is done with the help of figure renderer classes that have a class hierarchy roughly parallel to the `cFigure` inheritance tree. The base classes are incidentally called `FigureRenderer`. How figure renderers do their job may be different in various graphical runtime interfaces; in Qtenv, they create and manipulate `QGraphicsItem`s on a `QGraphicsView`. To be able to render a new figure type, one needs to create the appropriate figure renderer classes for Qtenv.

The names of the renderer classes are provided by the figures themselves, by their `getRendererClassName()` methods. For example, `cLineFigure`'s `getRendererClassName()` returns `LineFigureRenderer`. Qtenv qualifies that with its own namespace, and looks for a registered class named `omnetpp::qtenv::LineFigureRenderer`. If such class exists and is a Qtenv figure renderer (the appropriate `dynamic_cast` succeeds), an instance of that class will be used to render the figure, otherwise an error message will be issued.

8.7 3D Visualization

8.7.1 Introduction

OMNEST lets one build advanced 3D visualization for simulation models. 3D visualization is useful for wide range of simulations, including mobile wireless networks, transportation models, factory floorplan simulations and more. One can visualize terrain, roads, urban street networks, indoor environments, satellites, and more. It is possible to augment the 3D scene with various annotations. For wireless network simulations, for example, one can create a scene that, in addition to the faithful representation of the physical world, also displays the transmission range of wireless nodes, their connectivity graph and various statistics, indicates individual wireless transmissions or traffic intensity, and so on.

In OMNEST, 3D visualization is completely distinct from display string-based and canvas-based visualization. The scene appears on a separate GUI area.

OMNEST's 3D visualization is based on the open-source OpenSceneGraph and osgEarth libraries. These libraries offer high-level functionality, such as the ability of using 3D model files directly, accessing and rendering online map and satellite imagery data sources, and so on.

OpenSceneGraph and osgEarth

OpenSceneGraph (openscenegraph.org), or OSG for short, is the base library. It is best to quote their web site:

> "OpenSceneGraph is an open source high performance 3D graphics toolkit, used by application developers in fields such as visual simulation, games, virtual reality, scientific visualization and modeling. Written entirely in Standard C++ and OpenGL, it runs on all Windows platforms, OS X, GNU/Linux, IRIX, Solaris, HP-UX, AIX and FreeBSD operating systems. OpenSceneGraph is now well established as the world leading scene graph technology, used widely in the vis-sim, space, scientific, oil-gas, games and virtual reality industries."

In turn, osgEarth (osgearth.org) is a geospatial SDK and terrain engine built on top of OpenSceneGraph, not quite unlike Google Earth. It has many attractive features:

- Able to use various street map providers, satellite imaging providers, elevation data sources, both online and offline
- Data from online sources may be exported into a file suitable for offline use
- Scene may be annotated with various types of graphical objects
- Includes conversion between various geographical coordinate systems

In OMNEST, osgEarth can be very useful for simulations involving maps, terrain, or satellites.

8.7.2 The OMNEST API for OpenSceneGraph

For 3D visualization, OMNEST basically exposes the OpenSceneGraph API. One needs to assemble an OSG scene graph in the model, and give it to OMNEST for display. The scene graph can be updated at runtime, and changes will be reflected in the display.

> **NOTE: What is a scene graph?** A scene graph is a tree-like directed graph data structure that describes a 3D scene. The root node represents the whole virtual world. The world is then broken down into a hierarchy of nodes representing either spatial groupings of objects, settings of the position of objects, animations of objects, or definitions of logical relationships between objects. The leaves of the graph represent the physical objects themselves, the drawable geometry and their material properties.

When a scene graph has been built by the simulation model, it needs to be given to a `cOsgCanvas` object to let the OMNEST GUI know about it. `cOsgCanvas` wraps a scene graph, plus hints for the GUI on how to best display the scene, for example the default camera position. In the GUI, the user can use the mouse to manipulate the camera to view the scene from various angles and distances, look at various parts of the scene, and so on.

It is important to note that the simulation model may only manipulate the scene graph, but it cannot directly access the viewer in the GUI. There is a very specific technical reason for that. The viewer may not even exist or may be displaying a different scene graph at the time the model tries to access it. The model may even be running under a non-GUI user interface (e.g. Cmdenv) where a viewer is not even part of the program. The viewer may only be influenced in the form of viewer hints in `cOsgCanvas`.

Creating and Accessing cOsgCanvas Objects

Every module has a built-in (default) `cOsgCanvas`, which can be accessed with the `getOsgCanvas()` method of `cModule`. For example, a toplevel submodule can get hold of the network's OSG canvas with the following line:

```
cOsgCanvas *osgCanvas = getParentModule()->getOsgCanvas();
```

Additional `cOsgCanvas` instances may be created simply with `new`:

```
cOsgCanvas *osgCanvas = new cOsgCanvas("scene2");
```

cOsgCanvas and Scene Graphs

Once a scene graph has been assembled, it can be set on `cOsgCanvas` with the `setScene()` method.

```
osg::Node *scene = ...
osgCanvas->setScene(scene);
```

Subsequent changes in the scene graph will be automatically reflected in the visualization, there is no need to call `setScene()` again or otherwise let OMNEST know about the changes.

Viewer Hints

There are several hints that the 3D viewer may take into account when displaying the scene graph. Note that hints are only hints, so the viewer may choose to ignore them, and the user may also be able to override them interactively, (using the mouse, via the context menu, hotkeys or by other means).

- **Viewer style.** The viewer style can be set with `setViewerStyle()` and it determines the default hints for a scene. Choices are `STYLE_GENERIC` that should be set for generic

(non-osgEarth) scenes (default), and `STYLE_EARTH` for osgEarth scenes. As a rule of thumb, `STYLE_EARTH` should be used only when the model is loading `.earth` files.

- **Camera manipulators.** The OSG viewer makes use of camera manipulators that map mouse and keyboard gestures to camera movement. Use `setCameraManipulatorType()` to specify a manipulator. Several camera manipulators are available: `CAM_TERRAIN` is suitable for flying above an object or terrain; `CAM_OVERVIEW` which is similar to the terrain manipulator, but does not allow rolling or looking up (one can only see the object from above); `CAM_TRACKBALL` that allows unrestricted movement centered around an object; and `CAM_EARTH` that should be used when viewing the whole Earth is useful (i.e. modeling satellites). The default setting is to choose the manipulator automatically (`CAM_AUTO`) based on the viewer style (`CAM_OVERVIEW` or `CAM_EARTH`).

- **Scene rendering.** One can set the default background color for non-osgEarth scenes using `setClearColor()`. It is also possible to set the distances of the near and far clipping planes (`setZNear()` and `setZFar()`). Everything in the scene will be truncated to fit between these two planes. If you see parts of objects being clipped away from the scene, try to adjust these values. [6]

- **Viewpoint and field of view.** Default viewpoints can be set by `setGenericViewpoint(cOsgCanvas::Viewpoint&)` by specifying the x, y, z coordinates of the camera, the focal point and the "up" direction. For osgEarth scenarios, `setEarthViewpoint(osgEarth::Viewpoint&)` can be used to set the location of the observer and focal point using geographic coordinates. It is also possible to set the camera's field of view angle, with `setFieldOfViewAngle()`.

An example code fragment that sets some viewer hints:

```
osgCanvas->setViewerStyle(cOsgCanvas::STYLE_GENERIC);
osgCanvas->setCameraManipulatorType(cOsgCanvas::CAM_OVERVIEW);
osgCanvas->setClearColor(cOsgCanvas::Color("skyblue"));
osgCanvas->setGenericViewpoint(cOsgCanvas::Viewpoint(
        cOsgCanvas::Vec3d(20, -30, 30), // observer
        cOsgCanvas::Vec3d(30, 20, 0),   // focal point
        cOsgCanvas::Vec3d(0, 0, 1)));   // UP
```

Making Nodes Selectable

If a 3D object in the scene represents a C++ object in the simulation, it would often be very convenient to be able to select that object for inspection by clicking it with the mouse.

OMNEST provides a wrapper node that associates its children with a particular OMNEST object (`cObject` descendant), making them selectable in the 3D viewer. The wrapper class is called `cObjectOsgNode`, and it subclasses from `osg::Group`.

[6] OSG renders the scene using a *Z-buffer*. This means that upon drawing, the distance of every pixel of every object from the camera (called its depth) will be compared to the distance of the last drawn pixel in the same position, which is stored in the Z-buffer. The pixel will only be updated with the new color if it is found to be closer than the previous. Using a Z-buffer simplifies the rendering process, but the limited precision of the depth values will cause some pixels to be considered equidistant from the camera even if they are not. In this case, the result of the comparison, and thus the final color of the pixel is undefined, causing visual glitches called *Z-fighting* (flashing objects). $zNear$ and $zFar$ should be chosen such that no important objects are left out of the rendering, and in the same time Z-fighting is minimized. As a rule of thumb, the $zFar/zNear$ ratio should not exceed about 10,000, regardless of their absolute value.

```
auto objectNode = new cObjectOsgNode(myModule);
objectNode->addChild(myNode);
```

NOTE: The OMNEST object should exist as long as the wrapper node exists. Otherwise, clicking child nodes with the mouse is likely to result in a crash.

Finding Resources

3D visualizations often need to load external resources from disk, for example images or 3D models. By default, OSG tries to load these files from the current working directory (unless they are given with absolute path). However, loading from the folder of the current OMNEST module, from the folder of the ini file, or from the image path would often be more convenient. OMNEST contains a function for that purpose.

The `resolveResourcePath()` method of modules and channels accepts a file name (or relative path) as input, and looks into a number of convenient locations to find the file. The list of the search folders includes the current working directory, the folder of the main ini file, and the folder of the NED file that defined the module or channel. If the resource is found, the function returns the full path; otherwise it returns the empty string.

The function also looks into folders on the NED path and the image path, i.e. the roots of the NED and image folder trees. These search locations allow one to load files by full NED package name (but using slashes instead of dots), or access an icon with its full name (e.g. `block/sink`).

An example that attempts to load a `car.osgb` model file:

```
std::string fileLoc = resolveResourcePath("car.osgb");
if (fileLoc == "")
    throw cRuntimeError("car.osgb not found");
auto node = osgDB::readNodeFile(fileLoc); // use the resolved path
```

Conditional Compilation

OSG and osgEarth are optional in OMNEST, and may not be available in all installations. However, one probably wants simulation models to compile even if the particular OMNEST installation doesn't contain the OSG and osgEarth libraries. This can be achieved by conditional compilation.

OMNEST detects the OSG and osgEarth libraries and defines the `WITH_OSG` macro if they are present. OSG-specific code needs to be surrounded with `#ifdef WITH_OSG`.

An example:

```
...
#ifdef WITH_OSG
#include <osgDB/ReadFile>
#endif

void DemoModule::initialize()
{
#ifdef WITH_OSG
    cOsgCanvas *osgCanvas = getParentModule()->getOsgCanvas();
```

```
    osg::Node *scene = ... // assemble scene graph here
    osgCanvas->setScene(scene);
    osgCanvas->setClearColor(cOsgCanvas::Color(0,0,64)); // hint
#endif
}
```

Using Additional Libraries

OSG and osgEarth are comprised of several libraries. By default, OMNEST links simulations with only a subset of them: `osg`, `osgGA`, `osgViewer`, `osgQt`, `osgEarth`, `osgEarthUtil`. When additional OSG and osgEarth libraries are needed, one needs to ensure that those libraries are linked to the model as well. The best way to achieve that is to use the following code fragment in the `makefrag` file of the project:

```
ifneq ($(OSG_LIBS),)
LIBS += $(OSG_LIBS) -losgDB -losgAnimation ... # additional OSG libs
endif
ifneq ($(OSGEARTH_LIBS),)
LIBS += $(OSGEARTH_LIBS) -losgEarthFeatures -losgEarthSymbology ...
endif
```

The `ifneq()` statements ensure that `LIBS` is only updated if OMNEST has detected the presence of OSG/osgEarth in the first place.

8.7.3 Using OSG

OpenScenegraph is a sizable library with 16+ namespaces and 40+ `osg::Node` subclasses, and we cannot fully document it here due to size constraints. Instead, in the next sections we have collected some practical advice and useful code snippets to help the reader get started. More information can be found on the openscenegraph.org web site, in dedicated OpenSceneGraph books (some of which are freely available), and in other online resources. We list some OSG-related resources at the end of this chapter.

Loading Models

To display a 3D model in the canvas of a compound module, an `osg::Node` has to be provided as the root of the scene.

One method of getting such a `Node` is to load it from a file containing the model. This can be done with the `osgDB::readNodeFile()` method (or with one of its variants). This method takes a string as argument, and based on the protocol specification and extension(s), finds a suitable loader for it, loads it, finally returns with a pointer to the newly created `osg::Node` instance.

This node can now be set on the canvas for display with the `setScene()` method, as seen in the osg-intro sample (among others):

```
osg::Node *model = osgDB::readNodeFile("model.osgb");
getParentModule()->getOsgCanvas()->setScene(model);
```

NOTE: Where to get model files? While OpenSceneGraph recognizes and can load a wide range of formats, many 3D modeling tools can also export the edited scene or part of it in OSG's native file format, osgt, with the help of exporter plugins. One such plugin for Blender has been used to develop some of the OSG demos for OMNEST, and it was found to be working well.

There is support for so-called "pseudo loaders" in OSG, which provide additional options for loading models. Those allow for some basic operations to be performed on the model after it is loaded. To use them, simply append the parameters for the modifier followed by the name of it to the end of the file name upon loading the model.

Take this line from the osg-earth sample for example:

```
*.cow[*].modelURL = "cow.osgb.2.scale.0,0,90.rot.0,0,-15e-1.trans"
```

This string will first scale the original cow model in `cow.osgb` to 200%, then rotate it 90 degrees around the Z axis and finally translate it 1.5 units downwards. The floating point numbers have to be represented in scientific notation to avoid the usage of decimal points or commas in the number as those are already used as operator and parameter separators.

Note that these modifiers operate directly on the model data and are independent of any further dynamic transformations applied to the node when it is placed in the scene. For further information refer to the OSG knowledge base.

Creating Shapes

Shapes can also be built programatically. For that, one needs to use the `osg::Geode`, `osg::ShapeDrawable` and `osg::Shape` classes.

To create a shape, one first needs to create an `osg::Shape`. `osg::Shape` is an abstract class and it has several subclasses, like `osg::Box`, `osg::Sphere`, `osg::Cone`, `osg::Cylinder` or `osg::Capsule`. That object is only an abstract definition of the shape, and cannot be drawn on its own. To make it drawable, one needs to create an `osg::ShapeDrawable` for it. However, an `osg::ShapeDrawable` still cannot be attached to the scene, as it is still not an `osg::Node`. The `osg::ShapeDrawable` must be added to an `osg::Geode` (*geometry node*) to be able to insert it into the scene. This object can then be added to the scene and positioned and oriented freely, just like any other `osg::Node`.

For an example of this see the following snippet from the osg-satellites sample. This code creates an `osg::Cone` and adds it to the scene.

```
auto cone = new osg::Cone(osg::Vec3(0, 0, -coneRadius*0.75),
                          coneHeight, coneRadius);
auto coneDrawable = new osg::ShapeDrawable(cone);
auto coneGeode = new osg::Geode;
coneGeode->addDrawable(coneDrawable);
locatorNode->addChild(coneGeode);
```

Note that a single `ost::Shape` instance can be used to construct many `osg::ShapeDrawable`s, and a single `osg::ShapeDrawable` can be added to any number of `osg::Geode`s to make it appear in multiple places or sizes in the scene. This can in fact improve rendering performance.

Placing and Orienting Models in a Scene

One way to position and orient nodes is by making them children of an `osg::PositionAttitudeTransform`. This node provides methods to set the position, orientation and scale of its children. Orientation is done with quaternions (`osg::Quat`). Quaternions can be constructed from an axis of rotation and a rotation angle around the axis.

The following example places a node at the (x, y, z) coordinates and rotates it around the Z axis by `heading` radians to make it point in the right direction.

```
osg::Node *objectNode = ...;
auto transformNode = new osg::PositionAttitudeTransform();
transformNode->addChild(objectNode);
transformNode->setPosition(osg::Vec3d(x, y, z));
double heading = ...; // (in radians)
transformNode->setAttitude(osg::Quat(heading, osg::Vec3d(0, 0, 1)));
```

Adding Labels and Annotations

OSG makes it possible to display text or image labels in the scene. Labels are rotated to be always parallel to the screen, and scaled to appear in a constant size. In the following we'll show an example where we create a label and display it relative to an arbitrary node.

First, the label has to be created:

```
auto label = new osgText::Text();
label->setCharacterSize(18);
label->setBoundingBoxColor(osg::Vec4(1.0, 1.0, 1.0, 0.5)); // RGBA
label->setColor(osg::Vec4(0.0, 0.0, 0.0, 1.0)); // RGBA
label->setAlignment(osgText::Text::CENTER_BOTTOM);
label->setText("Hello World");
label->setDrawMode(osgText::Text::FILLEDBOUNDINGBOX | osgText::Text::TEXT);
```

Or if desired, a textured rectangle with an image:

```
auto image = osgDB::readImageFile("myicon.png");
auto texture = new osg::Texture2D();
texture->setImage(image);
auto icon = osg::createTexturedQuadGeometry(osg::Vec3(0.0, 0.0, 0.0),
    osg::Vec3(image->s(), 0.0, 0.0), osg::Vec3(0.0, image->t(), 0.0),
    0.0, 0.0, 1.0, 1.0);
icon->getOrCreateStateSet()->setTextureAttributeAndModes(0, texture);
icon->getOrCreateStateSet()->setMode(GL_DEPTH_TEST, osg::StateAttribute::ON);
```

If the image has transparent parts, one also needs the following lines:[7]

```
icon->getOrCreateStateSet()->setMode(GL_BLEND, osg::StateAttribute::ON);
icon->getOrCreateStateSet()->setRenderingHint(osg::StateSet::TRANSPARENT_BIN);
```

The icon and/or label needs an `osg::Geode` to be placed in the scene. Lighting is best disabled for the label.

[7] These lines enable blending, and places `icon` in the `TRANSPARENT_BIN`. Normally there are two bins, *opaque* and *transparent*. When a scene is rendered, OSG first renders the objects in the opaque bin, then the objects in the transparent bin. More bits can be created, but that is rarely necessary.

```
auto geode = new osg::Geode();
geode->getOrCreateStateSet()->setMode(GL_LIGHTING,
            osg::StateAttribute::OFF | osg::StateAttribute::OVERRIDE);
double labelSpacing = 2;
label->setPosition(osg::Vec3(0.0, labelSpacing, 0.0));
geode->addDrawable(label);
geode->addDrawable(icon);
```

This `osg::Geode` should be made a child of an `osg::AutoTransform` node, which applies the correct transformations to it for the label-like behaviour to happen:

```
auto autoTransform = new osg::AutoTransform();
autoTransform->setAutoScaleToScreen(true);
autoTransform->setAutoRotateMode(osg::AutoTransform::ROTATE_TO_SCREEN);
autoTransform->addChild(geode);
```

This `autoTransform` can now be made a child of the `modelToTransform`, and moved with it.Alternatively, both can be added to a new `osg::Group`, as siblings, and handled together using that.

We want the label to appear relative to an object called `modelNode`. One way would be to make `autoTransform` the child of `modelNode`, but here we rather place both of them under an `osg::Group`. The group should be inserted

```
auto modelNode = ... ;
auto group = new osg::Group();
group->addChild(modelNode);
group->addChild(autoTransform);
```

To place the label above the object, we set its position to $(0, 0, z)$, where z is the radius of the object's bounding sphere.

```
auto boundingSphere = modelNode->getBound();
autoTransform->setPosition(osg::Vec3d(0.0, 0.0, boundingSphere.radius()));
```

Drawing Lines

To draw a line between two points in the scene, first the two points have to be added into an `osg::Vec3Array`. Then an `osg::DrawArrays` should be created to specify which part of the array needs to be drawn. In this case, it is obviously two points, starting from the one at index 0. Finally, an `osg::Geometry` is necessary to join the two together.

```
auto vertices = new osg::Vec3Array();
vertices->push_back(osg::Vec3(begin_x, begin_y, begin_z));
vertices->push_back(osg::Vec3(end_x, end_y, end_z));

auto drawArrays = new osg::DrawArrays(osg::PrimitiveSet::LINE_STRIP);
drawArrays->setFirst(0);
drawArrays->setCount(vertices->size());

auto geometry = new osg::Geometry();
geometry->setVertexArray(vertices);
geometry->addPrimitiveSet(drawArrays);
```

The resulting `osg::Geometry` must be added to an `osg::Geode` (*geometry node*), which makes it possible to add it to the scene.

```
auto geode = new osg::Geode();
geode->addDrawable(geometry);
```

To change some visual properties of the line, the `osg::StateSet` of the `osg::Geode` has to be modified. The width of the line, for example, is controlled by a `osg::StateAttribute` called `osg::LineWidth`.

```
float width = ...;
auto stateSet = geode->getOrCreateStateSet();
auto lineWidth = new osg::LineWidth();
lineWidth->setWidth(width);
stateSet->setAttributeAndModes(lineWidth, osg::StateAttribute::ON);
```

Because of how `osg::Geometry` is rendered, the specified line width will always be constant on the screen (measured in pixels), and will not vary based on the distance from the camera. To achieve that effect, a long and thin `osg::Cylinder` could be used instead.

Changing the color of the line can be achieved by setting an appropriate `osg::Material` on the `osg::StateSet`. It is recommended to disable lighting for the line, otherwise it might appear in a different color, depending on where it is viewed from or what was rendered just before it.[8]

```
auto material = new osg::Material();
osg::Vec4 colorVec(red, green, blue, opacity); // all between 0.0 and 1.0
material->setAmbient(Material::FRONT_AND_BACK, colorVec);
material->setDiffuse(Material::FRONT_AND_BACK, colorVec);
material->setAlpha(Material::FRONT_AND_BACK, opacity);
stateSet->setAttribute(material);
stateSet->setMode(GL_LIGHTING,
            osg::StateAttribute::OFF | osg::StateAttribute::OVERRIDE);
```

How to Organize a Scene

Independent of how the scene has been constructed, it is always important to keep track of how the individual nodes are related to each other in the scene graph. This is because every modification of an `osg::Node` is by default propagated to all of its children, let it be a transformation, a render state variable, or some other flag.

For really simple scenes it might be enough to have an `osg::Group` as the root node, and make every other object a direct child of that. This reduces the complications and avoids any strange surprises regarding state inheritance. For more complex scenes it is advisable to follow the logical hierarchy of the displayed objects in the scene graph.

Once the desired object has been created and added to the scene, it can be easily moved and oriented to represent the state of the simulation by making it a child of an `osg::PositionAttitudeTransform` node.

[8] Since no normals were specified for the vertices upon creation, they are undefined (and wouldn't make much sense for a one-dimensional object), but still would be used for lighting.

Using Animations

If the node loaded by `readNodeFile()` contains animations (sometimes called actions), the `osgAnimation` module is capable of playing them back.

In simple cases, when there is only a single animation, and it is set up to play in a loop automatically (like the walking man in the osg-indoor sample simulation), there is no need to explicitly control it (provided it is the desired behaviour.)

Otherwise, the individual actions can be controlled by an `osgAnimation::AnimationManager`, with methods like `playAnimation()`, `stopAnimation()`, `isPlaying()`, etc. Animation managers can be found among the descendants of the loaded `osg::Node`s which are animated, for example using a custom `osg::NodeVisitor`:

```
osg::Node *objectNode = osgDB::readNodeFile( ... );

struct AnimationManagerFinder : public osg::NodeVisitor {
    osgAnimation::BasicAnimationManager *result = nullptr;
    AnimationManagerFinder()
      : osg::NodeVisitor(osg::NodeVisitor::TRAVERSE_ALL_CHILDREN) {}
    void apply(osg::Node& node) {
        if (result) return; // already found it
        if (osgAnimation::AnimationManagerBase* b =
              dynamic_cast<osgAnimation::AnimationManagerBase*>(
                node.getUpdateCallback())) {
            result = new osgAnimation::BasicAnimationManager(*b);
            return;
        }
        traverse(node);
    }
} finder;

objectNode->accept(finder);
animationManager = finder.result;
```

This visitor simply finds the first node in the subtree which has an update callback of type `osgAnimation::AnimationManagerBase`. Its result is a new `osgAnimation::BasicAnimationManager` created from the base.

This new `animationManager` now has to be set as an update callback on the `objectNode` to be able to actually drive the animations. Then any animation in the list returned by `getAnimationList()` can be set up as needed and played.

```
objectNode->setUpdateCallback(animationManager);
auto animation = animationManager->getAnimationList().front();
animation->setPlayMode(osgAnimation::Animation::STAY);
animation->setDuration(2);
animationManager->playAnimation(animation);
```

State Sets

Every `osg::Drawable` can have an `osg::StateSet` attached to it. An easy way of accessing it is via the `getOrCreateStateSet()` method of the drawable node. An `osg::StateSet` encapsulates a subset of the OpenGL state, and can be used to modify various rendering

parameters, for example the used textures, shader programs and their parameters, color and material, face culling, depth and stencil options, and many more `osg::StateAttributes`.

The following example enables blending for a node and sets up a transparent, colored material to be used for rendering it, through its `osg::StateSet`.

```
auto stateSet = node->getOrCreateStateSet();
stateSet->setMode(GL_BLEND, osg::StateAttribute::ON);
auto matColor = osg::Vec4(red, green, blue, alpha); // all between 0.0 and 1.0
auto material = new osg::Material;
material->setEmission(osg::Material::FRONT, matColor);
material->setDiffuse(osg::Material::FRONT, matColor);
material->setAmbient(osg::Material::FRONT, matColor);
material->setAlpha(osg::Material::FRONT, alpha);
stateSet->setAttributeAndModes(material, osg::StateAttribute::OVERRIDE);
```

To help OSG with the correct rendering of objects with transparency, they should be placed in the `TRANSPARENT_BIN` by setting up a rendering hint on their `osg::StateSet`. This ensures that they will be drawn after all fully opaque objects, and in decreasing order of their distance from the camera. When there are multiple transparent objects intersecting each other in the scene (like the transmission "bubbles" in the BostonPark configuration of the osg-earth sample simulation), there is no order in which they would appear correctly. A solution for these cases is to disable writing to the depth buffer during their rendering using the `osg::Depth` attribute.

```
stateSet->setRenderingHint(osg::StateSet::TRANSPARENT_BIN);
osg::Depth* depth = new osg::Depth;
depth->setWriteMask(false);
stateSet->setAttributeAndModes(depth, osg::StateAttribute::ON);
```

Please note that this still does not guarentee a completely physically accurate look, since that is a much harder problem to solve, but at least minimizes the obvious visual artifacts. Also, too many transparent objects might decrease performance, so wildly overusing them is to be avoided.

8.7.4 Using osgEarth

osgEarth is a cross-platform terrain and mapping SDK built on top of OpenSceneGraph. The most visible feature of osgEarth is that it adds support for loading `.earth` files to `osgDB::readNodeFile()`. An `.earth` file specifies contents and appearance of the displayed globe. This can be as simple as a single image textured over a sphere or as complex as realistic terrain data and satellite images complete with street and building information dynamically streamed over the internet from a publicly available provider, thanks to the flexibility of osgEarth. osgEarth also defines additional APIs to help with coordinate conversions and other tasks. Other than that, one's OSG knowledge is also applicable when building osgEarth scenes.

The next sections contain some tips and code fragments to help the reader get started with osgEarth. As with OSG, there are numerous other sources of information, both printed and online, when the info contained herein is insufficient.

Earth Files

When the osgEarth plugin is used to display a map as the visual environment of the simulation, its appearance can be described in a .earth file.

It can be loaded using the `osgDB::readNodeFile()` method, just like any other regular model. The resulting `osg::Node` will contain a node with a type of `osgEarth::MapNode`, which can be easily found using the `osgEarth::MapNode::findMapNode()` function. This node serves as the data model that contains all the data specified in the `.earth` file.

```
auto earth = osgDB::readNodeFile("example.earth");
auto mapNode = osgEarth::MapNode::findMapNode(earth);
```

An .earth file can specify a wide variety of options. The `type` attribute of the `map` tag (which is always the root of the document) lets the user select whether the terrain should be projected onto a flat plane (`projected`), or rendered as a geoid (`geocentric`).

Where the texture of the terrain is acquired from is specified by `image` tags. Many different kinds of sources are supported, including local files and popular online map sources with open access like MapQuest or OpenStreetMap. These can display different kinds of graphics, like satellite imagery, street or terrain maps, or other features the given on-line service provides.

The following example .earth file will set up a spherical rendering of Earth with textures from openstreetmap.org:

```
<map name="OpenStreetMap" type="geocentric" version="2" >
    <image name="osm_mapnik" driver="xyz" >
        <url>http://[abc].tile.openstreetmap.org/{z}/{x}/{y}.png</url>
    </image>
</map>
```

Elevation data can also be acquired in a similarly simple fashion using the `elevation` tag. The next snippet demonstrates this:

```
<map name="readymap.org" type="geocentric" version="2" >
    <image name="readymap_imagery" driver="tms" >
        <url>http://readymap.org/readymap/tiles/1.0.0/7/</url>
    </image>
    <elevation name="readymap_elevation" driver="tms" >
        <url>http://readymap.org/readymap/tiles/1.0.0/9/</url>
    </elevation>
</map>
```

For a detailed description of the available image and elevation source drivers, refer to the online references of osgEarth, or use one of the sample .earth files shipped with it.

The following partial .earth file places a label over Los Angeles, an extruded ellipse (a hollow cylinder) next to it, and a big red flag nearby.

```
<map ... >
    ...
    <external>
        <annotations>
            <label text="Los Angeles" >
                <position lat="34.051" long="-117.974" alt="100" mode="relative"/>
            </label>
```

```
            <ellipse name="ellipse extruded" >
                <position lat="32.73" long="-119.0"/>
                <radius_major value="50" units="km"/>
                <radius_minor value="20" units="km"/>
                <style type="text/css" >
                    fill:             #ff7f007f;
                    stroke:           #ff0000ff;
                    extrusion-height: 5000;
                </style>
            </ellipse>

            <model name="flag model" >
                <url>flag.osg.18000.scale</url>
                <position lat="33" long="-117.75" hat="0"/>
            </model>
        </annotations>
    </external>
</map>
```

Creating Offline Tile Packages

Being able to use online map providers is very convenient, but it is often more desirable to use an offline map resource. Doing so not only makes the simulation usable without internet access, but also speeds up map loading and insulates the simulation against changes in the online environment (availablity, content and configuration changes of map servers).

There are two ways map data may come from the local disk: caching, and using a self-contained offline map package. In this section we'll cover the latter, and show how you can create an offline map package from online sources, using the command line tool called `osgearth_package`. The resulting package, unlike map cache, will also be redistributable.

Given the right arguments, `osgearth_package` will download the tiles that make up the map, and arrange them in a fairly standardized, self-contained package. It will also create a corresponding `.earth` file that can be later used just like any other.

For example, the osg-earth sample simulation uses a tile package which has been created with a command similar to this one:

```
$ osgearth_package --tms boston.earth --out offline-tiles \
    --bounds -71.0705566406 42.350425122434 -71.05957031 42.358543917497 \
    --max-level 18 --out-earth boston_offline.earth --mt --concurrency 8
```

The `--tms boston.earth` arguments mean that we want to create a package in TMS format from the input file boston.earth. The `--out offline-tiles` argument specifies the output directory.

The `--bounds` argument specifies the rectangle of the map we want to include in the package, in the order *xmin ymin xmax ymax* order, as standard WGS84 datum (longitude/latitude). These example coordinates include the Boston Common area, used in some samples. The size of this rectangle obviously has a big impact on the size of the resulting package.

The `--max-level 18` argument is the maximum level of detail to be saved. This is a simple way of adjusting the tradeoff between quality and required disk space. Values between 15 and 20 are generally suitable, depending on the size of the target area and the available storage capacity.

The `--out-earth boston_offline.earth` option tells the utility to generate an .earth file with the given name in the output directory that references the prepared tile package as image source.

The `--mt --concurrency 8` arguments will make the process run in multithreaded mode, using 8 threads, potentially speeding it up.

The tool has a few more options for controlling the image format and compression mode among others. Consult the documentation for details, or the short usage help accessible with the `-h` switch.

> **HINT:** There is also a GUI front-end for `osgearth_package`, called `osgearth_package_qt`. This tool provides an easy way to select the bounding rectangle on the actual map using the mouse, lets the user choose the input and output files and the export options, and performs the exporting, all without having to resort to a command line interface.

Placing Objects on a Map

To easily position a part of the scene together on a given geographical location, an `osgEarth::GeoTransform` is of great help. It takes geographical coordinates (longitude/latitude/altitude), and creates a simple Cartesian coordinate system centered on the given location, in which all of its children can be positioned painlessly, without having to worry about further coordinate transformations between Cartesian and geographic systems. To move and orient the children within this local system, `osg::PositionAttitudeTransform` can be used.

```
osgEarth::GeoTransform *geoTransform = new osgEarth::GeoTransform();
osg::PositionAttitudeTransform *localTransform = new osg::PositionAttitudeTransfor

mapNode->getModelLayerGroup()->addChild(geoTransform);
geoTransform->addChild(localTransform);
localTransform->addChild(objectNode);

geoTransform->setPosition(osgEarth::GeoPoint(mapNode->getMapSRS(), longitude, lati
localTransform->setAttitude(osg::Quat(heading, osg::Vec3d(0, 0, 1)));
```

Adding Annotations on a Map

To display additional information on top of the terrain, annotations can be used. These are special objects that can adapt to the shape of the surface. Annotations can be of many kinds, for example simple geometric shapes like circles, ellipses, rectangles, lines, polygons (which can be extruded upwards to make solids); texts or labels, arbitrary 3D models, or images projected onto the surface.

All the annotations that can be created declaratively from an .earth file, can also be programatically generated at runtime.

This example shows how the circular transmission ranges of the cows in the osg-earth sample are created in the form of a `osgEarth::Annotation::CircleNode` annotation. Some basic styling is applied to it using an `osgEarth::Style`, and the rendering technique to be used is specified.

```
auto scene = ...;
auto mapNode = osgEarth::MapNode::findMapNode(scene);
```

```
auto geoSRS = mapNode->getMapSRS()->getGeographicSRS();
osgEarth::Style rangeStyle;
rangeStyle.getOrCreate<PolygonSymbol>()->fill()->color() =
                                        osgEarth::Color(rangeColor);
rangeStyle.getOrCreate<AltitudeSymbol>()->clamping() =
                                        AltitudeSymbol::CLAMP_TO_TERRAIN;
rangeStyle.getOrCreate<AltitudeSymbol>()->technique() =
                                        AltitudeSymbol::TECHNIQUE_DRAPE;
rangeNode = new osgEarth::Annotation::CircleNode(mapNode.get(),
    osgEarth::GeoPoint:(geoSRS, longitude, latitude),
    osgEarth::Linear(radius, osgEarth::Units::METERS), rangeStyle);
mapNode->getModelLayerGroup()->addChild(rangeNode);
```

8.7.5 OpenSceneGraph/osgEarth Programming Resources

Online resources

Loading and manipulating OSG models:

- http://trac.openscenegraph.org/projects/osg/wiki/Support/UserGuides/Plugins
- http://trac.openscenegraph.org/projects/osg/wiki/Support/Tutorials/FileLoadingAndTransforms
- http://trac.openscenegraph.org/projects/osg/wiki/Support/KnowledgeBase/PseudoLoader

Creating 3D models for OpenSceneGraph using Blender:

- https://github.com/cedricpinson/osgexport

osgEarth online documentation:

- http://docs.osgearth.org/en/latest/references/earthfile.html
- http://docs.osgearth.org/en/latest/index.html

Sample code

Be sure to check the samples coming with the OpenSceneGraph installation, as they contain invaluable information.

- https://github.com/openscenegraph/osg/tree/master/examples
- https://github.com/openscenegraph/osg-data

Books

The following books can be useful for more complex visualization tasks:

- *OpenSceneGraph Quick Start Guide*, by Paul Martz.

  This book is a concise introduction to the OpenSceneGraph API. It can be purchased from http://www.osgbooks.com, and it is also available as a free pdf download.

- *OpenSceneGraph 3.0: Beginners Guide*, by Wang Rui. Packt Publishing, 2010.

  This book is a concise introduction to the main features of OpenSceneGraph which then leads the reader into the fundamentals of developing virtual reality applications. Practical instructions and explanations accompany every step.

- *OpenSceneGraph 3.0 Cookbook*, by Wang Rui and Qian Xuelei. Packt Publishing, 2010.

  This book contains 100 recipes in 9 chapters, focusing on different fields including the installation, nodes, geometries, camera manipulation, animations, effects, terrain building, data management, GUI integration.

Chapter 9

Building Simulation Programs

9.1 Overview

This chapter describes the process and tools for building executable simulation models from their source code.

As described in the the previous chapters, the source of an OMNEST model usually contains the following files:

- C++ (`.cc` and `.h`) files, containing simple module implementations and other code;
- Message (`.msg`) files, containing message definitions to be translated into C++ classes;
- NED (`.ned`) files with component declarations and topology descriptions;
- Configuration (`.ini`) files with model parameter assignments and other settings.

The process to turn the source into an executable form is this, in nutshell:

1. Message files are translated into C++ using the message compiler, `opp_msgc`
2. C++ sources are compiled into object form (`.o` files)
3. Object files are linked with the simulation kernel and other libraries to get an executable or a shared library

Note that apart from the first step, the process is the same as building any C/C++ program. Also note that NED and ini files do not play a part in this process, as they are loaded by the simulation program at runtime.

One needs to link with the following libraries:

- The simulation kernel and class library (the *oppsim* library) and its dependencies (*oppenvir*, *oppcommon*, *oppnedxml*, etc).
- Optionally, with one or more user interface libraries (*oppqtenv*, *oppcmdenv*). Note that these libraries themselves may depend on other libraries.

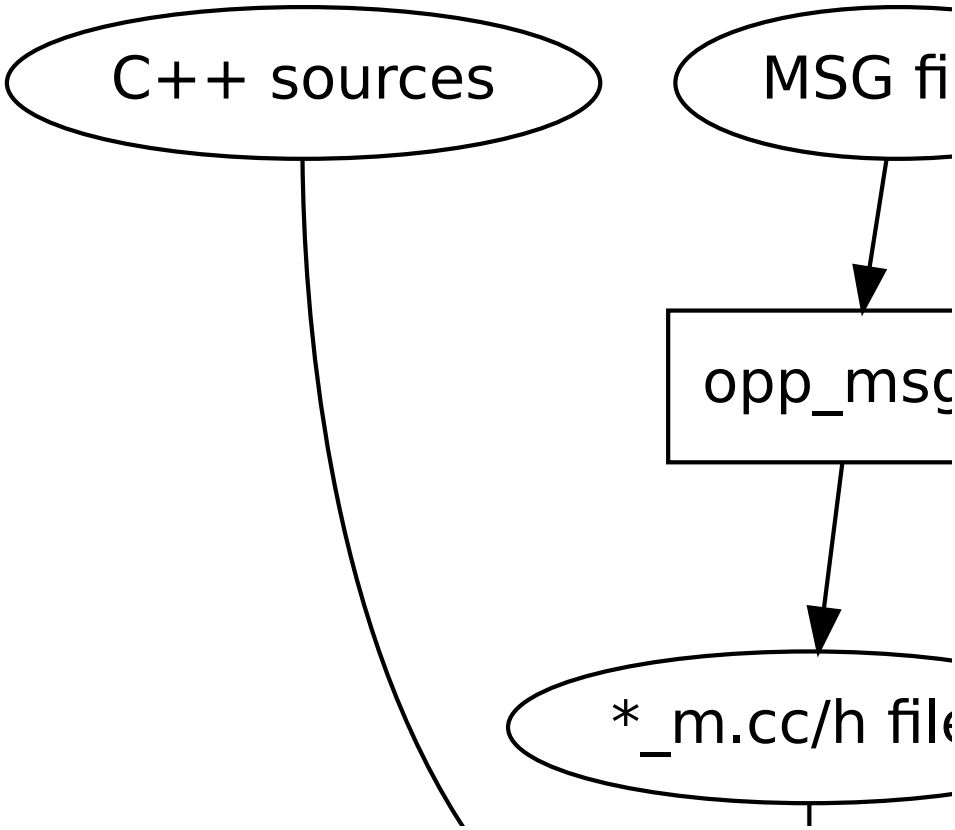


Figure 9.1: Building and running simulation

The exact files names of libraries depend on the platform and a number of additional factors.[1]

Figure 9.1 shows an overview of the process of building (and running) simulation programs.

You can see that the build process is not complicated. Tools such as `make` and `opp_makemake`, to be described in the rest of the chapter, are primarily needed to optimize rebuilds (if a message file has been translated already, there is no need to repeat the translation for every build unless the file has changed), and for automation.

9.2 Using opp_makemake and Makefiles

There are several tools available for managing the build of C/C++ programs. OMNEST uses the traditional way, Makefiles. Writing a Makefile is usually a tedious task. However, OMNEST provides a tool that can generate the Makefile for the user, saving manual labour.

`opp_makemake` can automatically generate a Makefile for simulation programs, based on the source files in the current directory and (optionally) in subdirectories.

[1] On Unix-like platforms, file names are prefixed with `lib`. For debug versions, a `d` is appended to the name. Static libraries have the `.a` suffix (except on Windows where the file extension is `.lib`). Shared libraries end in `.so` on Unix-like platforms (but `.dylib` on OS X), and `.dll` on Windows.

9.2.1 Command-line Options

The most important options accepted by `opp_makemake` are:

- `-f`, `--force` : Force overwriting existing Makefile
- `-o <filename>` : Name of simulation executable or library to be built.
- `-O <directory>`, `--out <directory>` : Specifies the name of the output directory tree for out-of-directory build
- `--deep` : Generates a "deep" Makefile. A deep Makefile will cover the whole source tree under the make directory, not just files in that directory.
- `-r`, `--recurse` : Causes make to recursively descend into all subdirectories; subdirectories are expected to contain Makefiles themselves.
- `-X <directory>`, `-X<directory>`, `--except <directory>` : With `-r` or `--deep`: ignore the given directory.
- `-d<subdir>`, `-d <subdir>`, `--subdir <subdir>` : Causes make to switch to the given directory and invoke a Makefile in that directory.
- `-n`, `--nolink` : Produce object files but do not create an executable or library.
- `-s`, `--make-so` : Build shared library (`.so`, `.dll` or `.dylib`).
- `-a`, `--make-lib` : Create static library (`.a` or `.lib`).
- `-I<dir>` : Add the given directory to the C++ include path.
- `-D<name>[=<value>]`, `-D <name>[=v<alue>]`, `-define <name>[=<value>]` : Define the given symbol in the C++ compiler.
- `-L<dir>` : Add the given directory to the library path.
- `-l<library>` : Additional library to link against.

There are several other options; run `opp_makemake -h` to see the complete list.

9.2.2 Basic Use

Assuming the source files (`*.ned`, `*.msg`, `*.cc`, `*.h`) are located in a single directory, one can change to that directory and type:

```
$ opp_makemake
```

This will create a file named `Makefile`. Now, running the `make` program will build a simulation executable.

```
$ make
```

IMPORTANT: The generated `Makefile` will contain the names of the sources files, so you need to re-run `opp_makemake` every time new files are added to or removed from the project.

To regenerate an existing `Makefile`, add the `-f` option to the command line, otherwise `opp_makemake` will refuse overwriting it.

```
$ opp_makemake -f
```

The name of the output file will be derived from the name of the project directory (see later). It can be overridden with the `-o` option:

```
$ opp_makemake -f -o aloha
```

The generated `Makefile` supports the following targets:

- `all` : Builds the simulation; this is also the default target.
- `clean` : Deletes files that were produced by the make process.

9.2.3 Debug and Release Builds

`opp_makemake` generates a Makefile that can create both release and debug builds. By default it creates release version, but it is easy to override this behavior by defining the `MODE` variable on the `make` command line.

```
$ make MODE=debug
```

It is also possible to generate a Makefile that defaults to debug builds. This can be achieved by adding the `--mode` option to the `opp_makemake` command line.

```
$ opp_makemake --mode debug
```

9.2.4 Debugging the Makefile

`opp_makemake` generates a Makefile that prints only minimal information during the build process (only the name of the compiled file.) To see the full compiler commands executed by the Makefile, add the `V=1` parameter to the `make` command line.

```
$ make V=1
```

9.2.5 Using External C/C++ Libraries

If the simulation model relies on an external library, the following `opp_makemake` options can be used to make the simulation link with the library.

- Use the `-I<dir>` option to specify the location of the header files. The directory will be added to the compiler's include path. This option is not needed if the header files are at a standard location, e.g. installed under `/usr/include` on Linux.
- Use the `-L<dir>` to specify the location of the binaries (static or shared library files.) Again, this option is not needed if the binaries are at a standard place, e.g. under `/usr/lib`.
- Use the `-l<libname>` to specify the name of the library. The name is normally the file name without the `lib` prefix and the file name extension (e.g. `.a`, `.so`, `.dylib`).

For example, linking with a hypothetical *Foo* library installed under `opt/` might require the following additional `opp_makemake` options: `-I/opt/foo/include -L/opt/foo/lib -lfoo`.

9.2.6 Building Directory Trees

It is possible to build a whole source directory tree with a single Makefile. A source tree will generate a single output file (executable or library). A source directory tree will always have a `Makefile` in its root, and source files may be placed anywhere in the tree.

To turn on this option, use the `opp_makemake --deep` option. `opp_makemake` will collect all `.cc` and `.msg` files from the whole subdirectory tree, and generate a Makefile that covers all. To exclude a specific directory, use the `-X exclude/dir/path` option. (Multiple `-X` options are accepted.)

An example:

```
$ opp_makemake -f --deep -X experimental -X obsolete
```

In the C++ code, include statements should contain the location of the file relative to the Makefile's location.[2] For example, if `Foo.h` is under `utils/common/` in the source tree, it needs to be included as

```
#include "utils/common/Foo.h"
```

9.2.7 Dependency Handling

The `make` program can utilize dependency information in the Makefile to shorten build times by omitting build steps whose input has not changed since the last build. Dependency information is automatically created and kept up-to-date during the build process.

Dependency information is kept in `.d` files in the output directory.

9.2.8 Out-of-Directory Build

The build system creates object and executable files in a separate directory, called the *output directory*. By default, the output directory is `out/<configname>`, where the `<configname>` part depends on the compiler toolchain and build mode settings. (For example, the result of a debug build with GCC will be placed in `out/gcc-debug`.) The subdirectory tree inside the output directory will mirror the source directory structure.

> **NOTE:** Generated source files (i.e. those created by `opp_msgc`) will be placed in the source tree rather than the output directory.

By default, the `out` directory is placed in the project root directory. This location can be changed with `opp_makemake`'s `-O` option.

```
$ opp_makemake -O ../tmp/obj
```

> **NOTE:** The project directory is identified as the first ancestor of the current directory that contains a `.project` file.

[2] Support for deep includes (automatically adding each subdirectory to the include path so that includes can be written without specifying the location of the file) has been dropped in OMNEST version 5.1, due to being error-prone in large projects, and having limited usefulness for small projects.

9.2.9 Building Shared and Static Libraries

By default the Makefile will create an executable file, but it is also possible to build shared or static libraries. Shared libraries are usually a better choice.

Use `--make-so` to create shared libraries, and `--make-lib` to build static libraries. The `--nolink` option completely omits the linking step, which is useful for top-level Makefiles that only invoke other Makefiles, or when custom linking commands are needed.

> **NOTE:** The Microsoft Visual C++ compiler handles shared library (DLL) linking differently, and requires additional steps to compile and link correctly. Namely, one must choose a `<symbol>` for the project, and annotate all classes, functions and variables to be exported from the DLL with `<symbol>_API`. Then, Makefiles need to be generated with the `-p <symbol>` option. This option will cause the `<symbol>_EXPORT` macro will be passed to the compiler, causing `<symbol>_API` to be defined as `__declspec(dllexport)` in the sources. This step allows Visual C++ to correctly generate DLL exports in Windows.

9.2.10 Recursive Builds

The `--recurse` option enables recursive make; when you build the simulation, make descends into the subdirectories and runs make in them too. By default, `--recurse` decends into all subdirectories; the `-X <dir>` option can be used to make it ignore certain subdirectories. This option is especially useful for top level Makefiles.

The `--recurse` option automatically discovers subdirectories, but this is sometimes inconvenient. Your source directory tree may contain parts which need their own hand written Makefile. This can happen if you include source files from an other non OMNEST project. With the `-d <dir>` or `--subdir <dir>` option, you can explicitly specify which directories to recurse into, and also, the directories need not be direct children of the current directory.

The recursive make options (`--recurse`, `-d`, `--subdir`) imply `-X`, that is, the directories recursed into will be automatically excluded from deep Makefiles.

You can control the order of traversal by adding dependencies into the `makefrag` file (see 9.2.11)

> **NOTE:** With `-d`, it is also possible to create infinite recursions. `opp_makemake` cannot detect them, it is your responsibility that cycles do not occur.

Motivation for recursive builds:

- toplevel Makefile
- integrating sources that have their own Makefile

9.2.11 Customizing the Makefile

It is possible to add rules or otherwise customize the generated Makefile by providing a `makefrag` file. When you run `opp_makemake`, it will automatically insert the content of the `makefrag` file into the resulting `Makefile`. With the `-i` option, you can also name other files to be included into the `Makefile`.

makefrag will be inserted after the definitions but before the first rule, so it is possible to override existing definitions and add new ones, and also to override the default target.

makefrag can be useful if some of your source files are generated from other files (for example, you use generated NED files), or you need additional targets in your Makefile or just simply want to override the default target in the Makefile.

NOTE: If you change the content of the makefrag file, you must recreate the Makefile using the opp_makemake command.

9.2.12 Projects with Multiple Source Trees

In the case of a large project, your source files may be spread across several directories and your project may generate more than one executable file (i.e. several shared libraries, examples etc.).

Once you have created your Makefiles with opp_makemake in every source directory tree, you will need a toplevel Makefile. The toplevel Makefile usually calls only the Makefiles recursively in the source directory trees.

9.2.13 A Multi-Directory Example

For a complex example of using opp_makemake, we will show how to create the Makefiles for a large project. First, take a look at the project's directory structure and find the directories that should be used as source trees:

```
project/
    doc/
    images/
    simulations/
    contrib/ <-- source tree (build libmfcontrib.so from this dir)
    core/ <-- source tree (build libmfcore.so from this dir)
    test/ <-- source tree (build testSuite executable from this dir)
```

Additionally, there are dependencies between these output files: mfcontrib requires mfcore and testSuite requires mfcontrib (and indirectly mfcore).

First, we create the Makefile for the core directory. The Makefile will build a shared lib from all .cc files in the core subtree, and will name it mfcore:

```
$ cd core && opp_makemake -f --deep --make-so -o mfcore -O out
```

The contrib directory depends on mfcore, so we use the -L and -l options to specify the library we should link with.

```
$ cd contrib && opp_makemake -f --deep --make-so -o mfcontrib -O out \
  -I../core -L../out/\$\(CONFIGNAME\)/core -lmfcore
```

The testSuite will be created as an executable file which depends on both mfcontrib and mfcore.

```
$ cd test && opp_makemake -f --deep -o testSuite -O out \
    -I../core -I../contrib -L../out/\$\(CONFIGNAME\)/contrib -lmfcontrib
```

Now, let us specify the dependencies among the above directories. Add the lines below to the `makefrag` file in the project root directory.

```
contrib_dir: core_dir
test_dir: contrib_dir
```

Now the last step is to create a top-level Makefile in the root of the project that calls the previously created Makefiles in the correct order. We will use the `--nolink` option, exclude every subdirectory from the build (`-X.`), and explicitly call the above Makefiles using `-d <dir>`. `opp_makemake` will automatically include the above created `makefrag` file.

```
$ opp_makemake -f --nolink -O out -d test -d core -d contrib -X.
```

9.3 Project Features

Long compile times are often an inconvenience when working with large OMNEST-based model frameworks. OMNEST has a facility named *project features* that lets you reduce build times by excluding or disabling parts of a large model library. For example, you can disable modules that you do not use for the current simulation study. 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 enforcing cleaner separation of unrelated parts in the model framework, being able to exclude code written for other platforms, and a less cluttered model palette in the NED editor.

NOTE: Modularization could also be achieved via breaking up the model framework into several smaller projects, but that would cause other kinds of inconveniences for model developers and users alike.

Project features can be enabled/disabled from both the IDE and the command line. It is possible to query the list of enabled project features, and use this information in creating a Makefile for the project.

9.3.1 What is a Project Feature

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

- Feature name; for example "UDP" or "Mobility examples".
- Feature description; This is a few sentences of text describing 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.
- NED packages; This is a 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 (nontypical), those folders are listed here.
- Compile options. When the feature is enabled, the compiler options listed here are added to the compiler command line of all C++ files in the project. Defines (`-D` options) are treated somewhat specially: the project can be set up so that defines go into a generated header file as `#define` lines instead of being added to the compiler command line. It is customary for each feature to have a corresponding symbol (`WITH_FOO` for a feature called *Foo*), so that other parts of the code can contain conditional blocks that are only compiled in when the given feature is enabled (or disabled).
- 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.

9.3.2 The opp_featuretool Program

Project features can be queried and manipulated using the `opp_featuretool` program. The first argument to the program must be a command; the most frequently used ones are `list`, `enable` and `disable`. The operation of commands can be refined with further options. One can obtain the full list of commands and options using the `-h` option.

Here are some examples of using the program.

Listing all features in the project:

```
$ opp_featuretool list
```

Listing all enabled features in the project:

```
$ opp_featuretool list -e
```

Enabling all features:

```
$ opp_featuretool enable all
```

Disabling a specific feature:

```
$ opp_featuretool disable Foo
```

The following command prints the command line options that should be used with `opp_makemake` to create a Makefile that builds the project with the currently enabled features:

```
$ opp_featuretool options
```

The easiest way to pass the output of the above command to `opp_makemake` is the `$(...)` shell construct:

```
$ opp_makemake --deep $(opp_featuretool options)
```

Often it is convenient to put feature defines (e.g. `WITH_FOO`) into a header file instead of passing them to the compiler via `-D` options. This makes it easier to detect feature enablements from derived projects, and also makes it easier for C++ code editors to correctly highlight conditional code blocks that depend on project features.

The header file can be generated with `opp_featuretool` using the following command:

```
$ opp_featuretool defines >feature_defines.h
```

At the same time, `-D` options must be removed from the compiler command line. `opp_featuretool options` has switches to filter them out. The modified command for Makefile generation:

```
$ opp_makemake --deep $(opp_featuretool options -fl)
```

It is advisable to create a Makefile rule that regenerates the header file when feature enablements change:

```
feature_defines.h: $(wildcard .oppfeaturestate) .oppfeatures
        opp_featuretool defines >feature_defines.h
```

9.3.3 The .oppfeatures File

Project features are defined in the `.oppfeatures` file in your project's root directory. This is an XML file, and it has to be written by hand (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. 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:

```
<feature
  id="TCP_common"
  name="TCP Common"
  description = "The common part of TCP implementations"
  initiallyEnabled = "true"
  requires = "IPv4"
  labels = "Transport"
  nedPackages = "inet.transport.tcp_common
                 inet.applications.tcpapp
                 inet.util.headerserializers.tcp"
  extraSourceFolders = ""
  compileFlags = "-DWITH_TCP_COMMON"
  linkerFlags = ""
  />
```

Project feature enablements are stored in the `.featurestate` file.

9.3.4 How to Introduce a Project Feature

If you plan to introduce a project feature in 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 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.
- Test. A rudimentary test is to verify that the project compiles at all, both with the new feature enabled and disabled. For projects with many features, automated build tests that compile the project using various feature configurations can be very useful. Such build tests can be written on top of `opp_featuretool`.

Chapter 10

Configuring Simulations

10.1 The Configuration File

Configuration and input data for the simulation are in a configuration file usually called `omnetpp.ini`.

10.1.1 An Example

For a start, let us see a simple `omnetpp.ini` file which can be used to run the Fifo example simulation.

```
[General]
network = FifoNet
sim-time-limit = 100h
cpu-time-limit = 300s
#debug-on-errors = true
#record-eventlog = true

[Config Fifo1]
description = "low job arrival rate"
**.gen.sendIaTime = exponential(0.2s)
**.gen.msgLength = 100b
**.fifo.bitsPerSec = 1000bps

[Config Fifo2]
description = "high job arrival rate"
**.gen.sendIaTime = exponential(0.01s)
**.gen.msgLength = 10b
**.fifo.bitsPerSec = 1000bps
```

The file is grouped into *sections* named `[General]`, `[Config Fifo1]` and `[Config Fifo2]`, each one containing several *entries*.

10.1.2 File Syntax

An OMNEST configuration file is a line-oriented text file. The encoding is primarily ASCII, but non-ASCII characters are permitted in comments and string literals. This allows for using encodings that are a superset of ASCII, for example ISO 8859-1 and UTF-8. There is no limit on the file size or on the line length.

Comments may be placed at the end of any line after a hash mark, "#". Comments extend to the end of the line, and are ignored during processing. Blank lines are also allowed and ignored.

Long lines can be broken to multiple lines in two ways: using the traditional *trailing backslash* notation also found in C/C++, or alternatively, by *indenting* the continuation lines.

When using the former method, the rule is that if the last character of a line is "\", it will be joined with the next line after removing the backslash and the newline. (Potential leading whitespace on the second line is preserved.) Note that this allows breaking the line even in the middle of a name, number or string constant.

When using latter method, a line can be broken between any two tokens by inserting a newline and indenting the next line. An indented line is interpreted as a continuation of the previous line. The first line and indented lines that follow it are then parsed as a single multi-line unit. Consequently, this method does not allow breaking a line in the middle of a word or inside string constants.

The two ways of breaking lines can be freely combined.

There are three types of lines: *section heading lines*, *key-value lines*, and *directive lines*:

1. *Section heading lines* contain a section name enclosed in square brackets.
2. *Key-value lines* have the <*key*>=<*value*> syntax; spaces are allowed (but not required) on both sides of the equal sign. If a line contains more than one equal sign, the leftmost one is taken as the key-value separator.
3. Currently there is only one kind of directive line, *include*. An include line starts with the `include` word, followed by the name of the file to be included.

Key-value lines may not occur above the first section heading line (except in included files, see later).

Keys may be further classified based on syntax alone:

1. Keys that do not contain dots represent global or per-run *configuration options*.
2. If a key contains a dot, its last component (substring after the last dot) is considered. If the last component contains a hyphen or is equal to `typename`, the key represents a *per-object configuration option*.
3. Otherwise, the key represents a *parameter assignment*. Thus, parameter assignment keys contain a dot, and no hyphen after the last dot.

An example:

```
# This is a comment line
[General]                       # section heading
network = Foo                   # configuration option
```

```
debug-on-errors = false          # another configuration option

**.vector-recording = false      # per-object configuration option
**.app*.typename = "HttpClient" # per-object configuration option

**.app*.interval = 3s            # parameter value
**.app*.requestURL = "http://www.example.com/this-is-a-very-very-very-very\
-very-long-url?q=123456789"      # a two-line parameter value
```

10.1.3 File Inclusion

OMNEST supports including an ini file in another, via the `include` keyword. This feature allows one to partition a large ini file into logical units, fixed and varying part, etc.

An example:

```
# omnetpp.ini
...
include params1.ini
include params2.ini
include ../common/config.ini
...
```

One can also include files from other directories. If the included ini file further includes others, their path names will be understood as relative to the location of the file which contains the reference, rather than relative to the current working directory of the simulation.

This rule also applies to other file names occurring in ini files (such as the **`load-libs`**, **`output-vector-file`**, **`output-scalar-file`**, etc. options, and `xmldoc()` module parameter values.)

In included files, it is allowed to have key-value lines without first having a section heading line. File inclusion is conceptually handled as text substitution, except that a section heading in an included file will not change the current section the main file. The following example illustrates the rules:

```
# incl.ini
foo1 = 1           # no preceding section heading: these lines will go into
foo2 = 2           # whichever section the file is included into
[Config Bar]
bar = 3            # this will always go to into [Config Bar]
```

```
# omnetpp.ini
[General]
include incl.ini  # adds foo1/foo2 to [General], and defines [Config Bar] w/ bar
baz1 = 4          # include files don't change the current section, so these
baz2 = 4          # lines still belong to [General]
```

NOTE: The concept of file inclusion implies that include files may not make sense on their own. Thus, when an included ini file is opened in the ini editor in the IDE, file contents may be flagged with errors and warnings. These errors/warnings disappear when the file is viewed as part of its main file.

10.2 Sections

An ini file may contain a `[General]` section, and several `[<configname>]` or `[Config <configname>]` sections. The use of the `Config` prefix is optional, i.e. `[Foo]` and `[Config Foo]` are equivalent.

The order of the sections is not significant.

10.2.1 The [General] Section

The most commonly used options of the `[General]` section are the following.

- The **`network`** option selects the model to be set up and run.
- The length of the simulation can be set with the **`sim-time-limit`** and the **`cpu-time-limit`** options (the usual time units such as ms, s, m, h, etc. can be used).

Note that the NED files loaded by the simulation may contain several networks, and any of them may be specified in the **`network`** option.

10.2.2 Named Configurations

Named configurations are in sections of the form `[Config <configname>]` or `[<configname>]` (the `Config` word is optional), where *<configname>* is by convention a camel-case string that starts with a capital letter: `Config1`, `WirelessPing`, `OverloadedFifo`, etc. For example, `omnetpp.ini` for an Aloha simulation might have the following skeleton:

```
[General]
...
[Config PureAloha]
...
[Config SlottedAloha1]
...
[Config SlottedAloha2]
...
```

Some configuration options (such as user interface selection) are only accepted in the `[General]` section, but most of them can go into `Config` sections as well.

When a simulation is run, one needs to select one of the configurations to be activated. In Cmdenv, this is done with the `-c` command-line option:

```
$ aloha -c PureAloha
```

The simulation will then use the contents of the `[Config PureAloha]` section to set up the simulation. (Qtenv, of course, lets the user choose the configuration from a dialog.)

10.2.3 Section Inheritance

When the PureAloha configuration is activated, the contents of the `[General]` section will also be taken into account: if some configuration option or parameter value is not found in `[Config PureAloha]`, then the search will continue in the `[General]` section. In other

words, lookups in `[Config PureAloha]` will fall back to `[General]`. The `[General]` section itself is optional; when it is absent, it is treated like an empty `[General]` section.

All named configurations fall back to `[General]` by default. However, for each configuration it is possible to specify the fall-back section or a list of fallback sections explicitly, using the **`extends`** key. Consider the following ini file skeleton:

```
[General]
...
[Config SlottedAlohaBase]
...
[Config LowTrafficSettings]
...
[Config HighTrafficSettings]
...

[Config SlottedAloha1]
extends = SlottedAlohaBase, LowTrafficSettings
...
[Config SlottedAloha2]
extends = SlottedAlohaBase, HighTrafficSettings
...
[Config SlottedAloha2a]
extends = SlottedAloha2
...
[Config SlottedAloha2b]
extends = SlottedAloha2
...
```

When `SlottedAloha2b` is activated, lookups will consider sections in the following order (this is also called the *section fallback chain*): `SlottedAloha2b`, `SlottedAloha2`, `SlottedAlohaBase`, `HighTrafficSettings`, `General`.

The effect is the same as if the contents of the sections SlottedAloha2b, SlottedAloha2, SlottedAlohaBase, HighTrafficSettings and General were copied together into one section, one after another, `[Config SlottedAloha2b]` being at the top, and `[General]` at the bottom. Lookups always start at the top, and stop at the first matching entry.

The order of the sections in the *fallback chain* is computed using the *C3 linearization algorithm* ([BCH+96]):

The *fallback chain* of a configuration `A` is

- if `A` does not have an **`extends`** key then `A`, `General`
- otherwise the merge of the configurations enumerated in the **`extends`** key, and all of their *fallback section chains*. The merge is *monotonic*: if some configuration `X` precedes configuration `Y` in one of the input chains, it will precede it in the output chain too.

The *section fallback chain* can be printed by the `-X` option of the command line of the simulation program:

```
$ aloha -X SlottedAloha2b
OMNEST Discrete Event Simulation
...
Config SlottedAloha2b
```

```
Config SlottedAloha2
Config SlottedAlohaBase
Config HighTrafficSettings
General
```

The *section fallback* concept is similar to multiple inheritance in object-oriented languages, and benefits are similar too; one can factor out the common parts of several configurations into a "base" configuration, and additionally, one can reuse existing configurations without copying, by using them as a base. In practice one will often have "abstract" configurations too (in the C++/Java sense), which assign only a subset of parameters and leave the others open, to be assigned in derived configurations.

When experimenting with a lot of different parameter settings for a simulation model, file inclusion and section inheritance can make it much easier to manage ini files.

10.3 Assigning Module Parameters

Simulations get input via module parameters, which can be assigned a value in NED files or in `omnetpp.ini` – in this order. Since parameters assigned in NED files cannot be overridden in `omnetpp.ini`, one can think about them as being "hardcoded". In contrast, it is easier and more flexible to maintain module parameter settings in `omnetpp.ini`.

In `omnetpp.ini`, module parameters are referred to by their full paths (hierarchical names). This name consists of the dot-separated list of the module names (from the top-level module down to the module containing the parameter), plus the parameter name (see section 7.1.2).

An example `omnetpp.ini` which sets the `numHosts` parameter of the toplevel module and the `transactionsPerSecond` parameter of the `server` module:

```
[General]
Network.numHosts = 15
Network.server.transactionsPerSecond = 100
```

Typename pattern assignments are also accepted:

```
[General]
Network.host[*].app.typename = "PingApp"
```

10.3.1 Using Wildcard Patterns

Models can have a large number of parameters to be configured, and it would be tedious to set them one-by-one in `omnetpp.ini`. OMNEST supports *wildcard patterns* which allow for setting several model parameters at once. The same pattern syntax is used for per-object configuration options; for example `<object-path-pattern>.record-scalar`, or `<module-path-pattern>.rng-<N>`.

The pattern syntax is a variation on Unix *glob*-style patterns. The most apparent differences to globbing rules are the distinction between `*` and `**`, and that character ranges should be written with curly braces instead of square brackets; that is, *any-letter* is expressed as `{a-zA-Z}` and not as `[a-zA-Z]`, because square brackets are reserved for the notation of module vector indices.

Pattern syntax:

- `?` : matches any character except dot (.)
- `*` : matches zero or more characters except dot (.)
- `**` : matches zero or more characters (any character)
- `{a-f}` : *set*: matches a character in the range a-f
- `{^a-f}`: *negated set*: matches a character NOT in the range a-f
- `{38..150}` : *numeric range*: any number (i.e. sequence of digits) in the range 38..150, inclusive; both limits are optional
- `[38..150]` : *index range*: any number in square brackets in the range 38..150, inclusive; both limits are optional
- backslash (\) : takes away the special meaning of the subsequent character

Precedence

The order of entries is very important with wildcards. When a key matches several wildcard patterns, the *first* matching occurrence is used. This means that one needs to list specific settings first, and more general ones later. Catch-all settings should come last.

An example ini file:

```
[General]
*.host[0].waitTime = 5ms    # specifics come first
*.host[3].waitTime = 6ms
*.host[*].waitTime = 10ms   # catch-all comes last
```

Asterisk vs Double Asterisk

The `*` wildcard is for matching a single module or parameter name in the path name, while `**` can be used to match several components in the path. For example, `**.queue*.bufSize` matches the `bufSize` parameter of any module whose name begins with `queue` in the model, while `*.queue*.bufSize` or `net.queue*.bufSize` selects only queues immediately on network level. Also note that `**.queue**.bufSize` would match `net.queue1.foo.bar.bufSize` as well!

Sets, Negated Sets

Sets and negated sets can contain several character ranges and also enumeration of characters. For example, `{_a-zA-Z0-9}` matches any letter or digit, plus the underscore; `{xyzc-f}` matches any of the characters x, y, z, c, d, e, f. To include '-' in the set, put it at a position where it cannot be interpreted as character range, for example: `{a-z-}` or `{-a-z}`. To include '}' in the set, it must be the first character: `{}a-z}`, or as a negated set: `{^}a-z}`. A backslash is always taken as a literal backslash (and not as an escape character) within set definitions.

Numeric Ranges and Index Ranges

Only nonnegative integers can be matched. The start or the end of the range (or both) can be omitted: `{10..}`, `{..99}` or `{..}` are valid numeric ranges (the last one matches any number). The specification must use exactly two dots. Caveat: `*{17..19}` will match `a17`, `117` and `963217` as well, because the `*` can also match digits!

An example for numeric ranges:

```
[General]
*.*.queue[3..5].bufSize = 10
*.*.queue[12..].bufSize = 18
*.*.queue[*].bufSize = 6  # this will only affect queues 0,1,2 and 6..11
```

10.3.2 Using the Default Values

It is also possible to utilize the default values specified in the NED files. The <*parameter-fullpath*>`=default` setting assigns the default value to a parameter if it has one.

The <*parameter-fullpath*>`=ask` setting will try to get the parameter value interactively from the user.

If a parameter was not set but has a default value, that value will be assigned. This is like having a `**=default` line at the bottom of the `[General]` section.

If a parameter was not set and has no default value, that will either cause an error or will be interactively prompted for, depending on the particular user interface.

> **NOTE:** In Cmdenv, one must explicitly enable interactive mode with the `--cmdenv-interactive=true` option, otherwise the simulation program will stop with an error in the setup phase.

More precisely, parameter resolution takes place as follows:

1. If the parameter is assigned in NED, it cannot be overridden in the configuration. The value is applied and the process finishes.

2. If the first match is a value line (matches <*parameter-fullpath*>=<*value*>), the value is applied and the process finishes.

3. If the first match is a <*parameter-fullpath*>`=default` line, the default value is applied and the process finishes.

4. If the first match is a <*parameter-fullpath*>`=ask` line, the parameter will be asked from the user interactively (UI dependent).

5. If there was no match and the parameter has a default value, it is applied and the process finishes.

6. Otherwise the parameter is declared unassigned, and handled accordingly by the user interface. It may be reported as an error, or may be asked from the user interactively.

10.4 Parameter Studies

It is quite common in simulation studies that the simulation model is run several times with different parameter settings, and the results are analyzed in relation to the input parameters. OMNEST 3.x had no direct support for batch runs, and users had to resort to writing shell (or Python, Ruby, etc.) scripts that iterated over the required parameter space, to generate a (partial) ini file and run the simulation program in each iteration.

OMNEST 4.x largely automates this process, and eliminates the need for writing batch execution scripts. It is the ini file where the user can specify iterations over various parameter settings. Here is an example:

```
[Config AlohaStudy]
*.numHosts = ${1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential(${0.2, 0.4, 0.6}s)
```

This parameter study expands to 8*3 = 24 simulation runs, where the number of hosts iterates over the numbers 1, 2, 5, 10, 20, 30, 40, 50, and for each host count three simulation runs will be done, with the generation interval being exponential(0.2), exponential(0.4), and exponential(0.6).

How can it be used? First of all, running the simulation program with the `-q numruns` option will print how many simulation runs a given configuration expands to.

```
$ ./aloha -c AlohaStudy -q numruns

OMNEST Discrete Event Simulation
...
Config: AlohaStudy
Number of runs: 24
```

When `-q runs` is used instead, the program will print the list of runs, with the values of the iteration variables for each run. (Use `-q rundetails` to get even more info.) Note that the parameter study actually maps to nested loops, with the last `${...}` becoming the innermost loop. The iteration variables are just named `$0` and `$1` – we'll see that it is possible to give meaningful names to them. Please ignore the `$repetition=0` part in the printout for now.

```
$ ./aloha -c AlohaStudy -q runs
OMNEST Discrete Event Simulation
...
Config: AlohaStudy
Number of runs: 24
Run 0: $0=1, $1=0.2, $repetition=0
Run 1: $0=1, $1=0.4, $repetition=0
Run 2: $0=1, $1=0.6, $repetition=0
Run 3: $0=2, $1=0.2, $repetition=0
Run 4: $0=2, $1=0.4, $repetition=0
Run 5: $0=2, $1=0.6, $repetition=0
Run 6: $0=5, $1=0.2, $repetition=0
Run 7: $0=5, $1=0.4, $repetition=0
...
Run 19: $0=40, $1=0.4, $repetition=0
Run 20: $0=40, $1=0.6, $repetition=0
Run 21: $0=50, $1=0.2, $repetition=0
Run 22: $0=50, $1=0.4, $repetition=0
```

```
Run 23: $0=50, $1=0.6, $repetition=0
```

Any of these runs can be executed by passing the `-r <runnumber>` option to Cmdenv. So, the task is now to run the simulation program 24 times, with `-r` running from 0 through 23:

```
$ ./aloha -u Cmdenv -c AlohaStudy -r 0
$ ./aloha -u Cmdenv -c AlohaStudy -r 1
$ ./aloha -u Cmdenv -c AlohaStudy -r 2
...
$ ./aloha -u Cmdenv -c AlohaStudy -r 23
```

This batch can be executed either from the OMNEST IDE (where you are prompted to pick an executable and an ini file, choose the configuration from a list, and just click Run), or using a little command-line batch execution tool (`opp_runall`) supplied with OMNEST.

Actually, it is also possible to make Cmdenv execute all runs in one go, by simply omitting the `-r` option.

```
$ ./aloha -u Cmdenv -c AlohaStudy

OMNEST Discrete Event Simulation
Preparing for running configuration AlohaStudy, run #0...
...
Preparing for running configuration AlohaStudy, run #1...
...
...
Preparing for running configuration AlohaStudy, run #23...
```

However, this approach is not recommended, because it is more susceptible to C++ programming errors in the model. (For example, if any of the runs crashes, the whole batch stops – which may not be what the user wants.)

10.4.1 Iterations

Let us return to the example ini file in the previous section:

```
[Config AlohaStudy]
*.numHosts = ${1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential( ${0.2, 0.4, 0.6}s )
```

The `${...}` syntax specifies an iteration. It is sort of a macro: at each run, the whole `${...}` string is textually replaced with the current iteration value. The values to iterate over do not need to be numbers (although the *"a..b"* and *"a..b step c"* forms only work on numbers), and the substitution takes place even inside string constants. So, the following examples are all valid (note that textual substitution is used):

```
*.param = 1 + ${1e-6, 1/3, sin(0.5)}
    ==> *.param = 1 + 1e-6
        *.param = 1 + 1/3
        *.param = 1 + sin(0.5)
*.greeting = "We will simulate ${1,2,5} host(s)."
    ==> *.greeting = "We will simulate 1 host(s)."
        *.greeting = "We will simulate 2 host(s)."
        *.greeting = "We will simulate 5 host(s)."
```

To write a literal `${..}` inside a string constant, quote the left brace with a backslash: `$\{..}`.

> **NOTE:** Inside `${..}`, the values are separated with commas. However, not every comma is taken as a value separator because the parser tries to be smart about what is meant. Commas inside (nested) parentheses, brackets or curly braces are ignored so that `${uniform(0,3)}` is parsed as one value and not as `uniform(0` plus `3)`. Commas, curly braces and other charachers inside double-quoted string literals are also ignored, so `${"Hello, world"}` yields a single `"Hello, world"` string and not `"Hello` plus `world"`. It is assumed that string literals use backslash as an escape characher, like in C/C++ and NED. To include a literal comma or close-brace inside a value, one needs to escape it with a backslash: `${foo\,bar\}baz}` will parse as a single value, `foo,bar}baz`. Backslashes themselves must be doubled. As the above examples illustrate, the parser removes one level of backslashes, except inside string literals where they are left intact.

10.4.2 Named Iteration Variables

One can assign names to iteration variables, which has the advantage that meaningful names will be displayed in the Cmdenv output instead of `$0` and `$1`, and also lets one reference iteration variables at other places in the ini file. The syntax is `${<varname>=<iteration>}`, and variables can be referred to simply as `${<varname>}`:

```
[Config Aloha]
*.numHosts = ${N=1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential( ${mean=0.2, 0.4, 0.6}s )
**.greeting = "There are ${N} hosts"
```

The scope of the variable name is the section that defines it, plus sections based on that section (via **`extends`**).

Referencing Other Iteration Variables

Iterations may refer to other iteration variables, using the dollar syntax (`$var`) or the dollar-brace syntax (`${var}`).

This feature makes it possible to have loops where the inner iteration range depends on the outer one. An example:

```
**.foo = ${i=1..10}  # outer loop
**.bar = ${j=1..$i}  # inner loop depends on $i
```

When needed, the default top-down nesting order of iteration loops is modified (loops are reordered) to ensure that expressions only refer to more outer loop variables, but not to inner ones. When this is not possible, an error is generated with the "circular dependency" message.

For instance, in the following example the loops will be nested in *k* - *i* - *j* order, *k* being the outermost and *j* the innermost loop:

```
**.foo = ${i=0..$k}   # must be inner to $k
**.bar = ${j=$i..$k}  # must be inner to both $i and $k
**.baz = ${k=1..10}   # may be the outermost loop
```

And the next example will stop with an error because there is no "good" ordering:

```
**.foo = ${i=0..$j}
**.bar = ${j=0..$k}
**.baz = ${k=0..$i} # --> error: circular references
```

Variables are substituted *textually*, and the result is normally *not* evaluated as an arithmetic expression. The result of the substitution is only evaluated where needed, namely in the three arguments of iteration ranges (*from*, *to*, *step*), and in the value of the **constraint** configuration option.

To illustrate textual substitution, consider the following contorted example:

```
**.foo = ${i=1..3, 1s+, -}001s
```

Here, the `foo` NED parameter will receiving the following values in subsequent runs: `1001s`, `2001s`, `3001s`, `1s+001s`, `-001s`.

CAUTION: Due to textual substitution, variables in arithmetic expressions should be protected with parentheses – just like in C/C++ function-style macros. Consider the following example:

```
**.foo = ${i=10}
**.bar = ${j=$i+5}
**.baz = ${k=2*$j}    # bogus! $j should be written as ($j)
constraint = $i+50 < 2*$j  # ditto: should use ($i) and ($j)
```

Here, the `baz` parameter will receive the string `2*10+5` after the substitutions and hence evaluate to 25 instead of the correct $2*(10+5)=30$; the constraint expression is similarly wrong. Mind the parens!

Substitution also works inside string constants within iterations (`${..}`).

```
**.foo = "${i=Jo}"  # -> Jo
**.bar = ${"Hi $i", "Hi ${i}hn"}  # -> Hi Jo /John
```

However, outside iterations the plain dollar syntax is not understood, only the dollar-brace syntax is:

```
**.foo = "${i=Day}"
**.baz = "Good $i"      # -> remains "Good $i"
**.baz = "Good ${i}"    # -> becomes "Good Day"
```

Rationale: The text substitution model was chosen for greater flexibility as well as the ability to produce more consistent semantics. The advantages outweigh the inconvenience of having to parenthesize variable references in arithmetic expressions.

10.4.3 Parallel Iteration

The body of an iteration may end in an exclamation mark followed by the name of another iteration variable. This syntax denotes a *parallel iteration*. A parallel iteration does not define a loop of its own, but rather, the sequence is advanced in lockstep with the variable after the "!". In other words, the "!" syntax chooses the *kth* value from the iteration, where *k* is the position (iteration count) of the iteration variable after the "!".

An example:

```
**.plan =     ${plan= "A", "B", "C", "D"}
**.numHosts = ${hosts= 10,  20,  50, 100 ! plan}
**.load =     ${load= 0.2, 0.3, 0.3, 0.4 ! plan}
```

In the above example, the only loop is defined by the first line, the `plan` variable. The other two iterations, `hosts` and `load` just follow it; for the first value of `plan` the first values of `host` and `load` are selected, and so on.

10.4.4 Predefined Variables, Run ID

There are a number of predefined variables: `${configname}` and `${runnumber}` with the obvious meanings; `${network}` is the name of the network that is simulated; `${processid}` and `${datetime}` expand to the OS process id of the simulation and the time it was started; and there are some more: `${runid}`, `${iterationvars}` and `${repetition}`.

`${runid}` holds the *run ID*. When a simulation is run, a run ID is assigned that uniquely identifies that instance of running the simulation: every subsequent run of the same simulation will produce a different run ID. The run ID is generated as the concatenation of several variables like `${configname}`, `${runnumber}`, `${datetime}` and `${processid}`. This yields an identifier that is unique "enough" for all practical purposes, yet it is meaningful for humans. The run ID is recorded into result files written during the simulation, and can be used to match vectors and scalars written by the same simulation run.

10.4.5 Constraint Expression

In cases when not all combinations of the iteration variables make sense or need to be simulated, it is possible to specify an additional constraint expression. This expression is interpreted as a conditional (an "if" statement) within the innermost loop, and it must evaluate to `true` for the variable combination to generate a run. The expression should be given with the **`constraint`** configuration option. An example:

```
*.numNodes = ${n=10..100 step 10}
**.numNeighbors = ${m=2..10 step 2}
constraint = ($m) <= sqrt($n)  # note: parens needed due to textual substitution
```

The expression syntax supports most C language operators including boolean, conditional and binary shift operations, and most `<math.h>` functions; data types are boolean, double and string. The expression must evaluate to a boolean.

> **NOTE:** Remember that variables are substituted textually into the expresssion, so they must be protected with parentheses to preserve evaluation order.

10.4.6 Repeating Runs with Different Seeds

It is directly supported to perform several runs with the same parameters but different random number seeds. There are two configuration options related to this: **`repeat`** and **`seed-set`**. The first one simply specifies how many times a run needs to be repeated. For example,

```
repeat = 10
```

causes every combination of iteration variables to be repeated 10 times, and the `${repetition}` predefined variable holds the loop counter. Indeed, `repeat=10` is equivalent to adding `${repetition=0..9}` to the ini file. The `${repetition}` loop always becomes the innermost loop.

The **seed-set** configuration key affects seed selection. Every simulation uses one or more random number generators (as configured by the **num-rngs** key), for which the simulation kernel can automatically generate seeds. The first simulation run may use one set of seeds (seed set 0), the second run may use a second set (seed set 1), and so on. Each set contains as many seeds as there are RNGs configured. All automatic seeds generate random number sequences that are far apart in the RNG's cycle, so they will never overlap during simulations.

> **NOTE:** Mersenne Twister, the default RNG of OMNEST has a cycle length of 2^{19937}, which is more than enough for any conceivable purpose.

The **seed-set** key tells the simulation kernel which seed set to use. It can be set to a concrete number (such as `seed-set=0`), but it usually does not make sense as it would cause every simulation to run with exactly the same seeds. It is more practical to set it to either `${runnumber}` or to `${repetition}`. The default setting is `${runnumber}`:

```
seed-set = ${runnumber}    # this is the default
```

This causes every simulation run to execute with a unique seed set. The second option is:

```
seed-set = ${repetition}
```

where all `$repetition=0` runs will use the same seeds (seed set 0), all `$repetition=1` runs use another seed set, `$repetition=2` a third seed set, etc.

To perform runs with manually selected seed sets, one needs to define an iteration for the **seed-set** key:

```
seed-set = ${5,6,8..11}
```

In this case, the **repeat** key should be left out, as **seed-set** already defines an iteration and there is no need for an extra loop.

It is of course also possible to manually specify individual seeds for simulations. The parallel iteration feature is very convenient here:

```
repeat = 4
seed-1-mt = ${53542, 45732, 47853, 33434 ! repetition}
seed-2-mt = ${75335, 35463, 24674, 56673 ! repetition}
seed-3-mt = ${34542, 67563, 96433, 23567 ! repetition}
```

The meaning of the above is this: in the first repetition, the first column of seeds is chosen, for the second repetition, the second column, etc. The "!" syntax chooses the *kth* value from the iteration, where *k* is the position (iteration count) of the iteration variable after the "!". Thus, the above example is equivalent to the following:

```
# no repeat= line!
seed-1-mt = ${seed1 = 53542, 45732, 47853, 33434}
seed-2-mt = ${         75335, 35463, 24674, 56673 ! seed1}
seed-3-mt = ${         34542, 67563, 96433, 23567 ! seed1}
```

That is, the iterators of `seed-2-mt` and `seed-3-mt` are advanced in lockstep with the `seed1` iteration.

10.4.7 Experiment-Measurement-Replication

We have introduced three concepts that are useful for organizing simulation results generated by batch executions or several batches of executions.

During a simulation study, a user prepares several *experiments*. The purpose of an experiment is to find out the answer to questions like *"how does the number of nodes affect response times in the network?"* For an experiment, several *measurements* are performed on the simulation model, and each measurement runs the simulation model with a different set of parameters. To eliminate the bias introduced by the particular random number stream used for the simulation, several *replications* of every measurement are run with different random number seeds, and the results are averaged.

OMNEST result analysis tools can take advantage of the *experiment*, *measurement* and *replication* labels recorded into result files, and display simulation runs and recorded results accordingly on the user interface.

These labels can be explicitly specified in the ini file using the **`experiment-label`**, **`measurement-label`** and **`replication-label`** config options. If they are missing, the default is the following:

```
experiment-label = "${configname}"
measurement-label = "${iterationvars}"
replication-label = "#${repetition},seed-set=<seedset>"
```

That is, the default experiment label is the configuration name; the measurement label is concatenated from the iteration variables; and the replication label contains the repeat loop variable and seed-set. Thus, for our first example the *experiment-measurement-replication* tree would look like this:

```
"PureAloha"--experiment
  $N=1,$mean=0.2 -- measurement
    #0, seed-set=0 -- replication
    #1, seed-set=1
    #2, seed-set=2
    #3, seed-set=3
    #4, seed-set=4
  $N=1,$mean=0.4
    #0, seed-set=5
    #1, seed-set=6
    ...
    #4, seed-set=9
  $N=1,$mean=0.6
    #0, seed-set=10
    #1, seed-set=11
    ...
    #4, seed-set=14
  $N=2,$mean=0.2
    ...
  $N=2,$mean=0.4
    ...
    ...
```

The *experiment-measurement-replication* labels should be enough to reproduce the same simulation results, given of course that the ini files and the model (NED files and C++ code)

haven't changed.

Every instance of running the simulation gets a unique run ID. We can illustrate this by listing the corresponding run IDs under each repetition in the tree. For example:

```
"PureAloha"
  $N=1,$mean=0.2
    #0, seed-set=0
      PureAloha-0-20070704-11:38:21-3241
      PureAloha-0-20070704-11:53:47-3884
      PureAloha-0-20070704-16:50:44-4612
    #1, seed-set=1
      PureAloha-1-20070704-16:50:55-4613
    #2, seed-set=2
      PureAloha-2-20070704-11:55:23-3892
      PureAloha-2-20070704-16:51:17-4615
      ...
```

The tree shows that ("PureAloha", "$N=1,$mean=0.2", "#0, seed-set=0") was run three times. The results produced by these three executions should be identical, unless, for example, some parameter was modified in the ini file, or a bug got fixed in the C++ code.

The default way of generating the *experiment/measurement/replication* labels is useful and sufficient for the majority of simulation studies. However, it can be customized if needed. For example, here is a way to join two configurations into one experiment:

```
[Config PureAloha_Part1]
experiment-label = "PureAloha"
...
[Config PureAloha_Part2]
experiment-label = "PureAloha"
...
```

Measurement and replication labels can be customized in a similar way, making use of named iteration variables, `${repetition}`, `${runnumber}` and other predefined variables. One possible benefit is to customize the generated measurement and replication labels. For example:

```
[Config PureAloha_Part1]
measurement = "${N} hosts, exponential(${mean}) packet generation interval"
```

One should be careful with the above technique though, because if some iteration variables are left out of the measurement labels, runs with all values of those variables will be grouped together to the same replications.

10.5 Configuring the Random Number Generators

The random number architecture of OMNEST was already outlined in section 7.3. Here we'll cover the configuration of RNGs in `omnetpp.ini`.

10.5.1 Number of RNGs

The **`num-rngs`** configuration option sets the number of random number generator instances (i.e. random number streams) available for the simulation model (see 7.3). Referencing an

RNG number greater or equal to this number (from a simple module or NED file) will cause a runtime error.

10.5.2 RNG Choice

The **rng-class** configuration option sets the random number generator class to be used. It defaults to "cMersenneTwister", the Mersenne Twister RNG. Other available classes are "cLCG32" (the "legacy" RNG of OMNEST 2.3 and earlier versions, with a cycle length of $2^{31}-2$), and "cAkaroaRNG" (Akaroa's random number generator, see section 11.20).

10.5.3 RNG Mapping

The RNG numbers used in simple modules may be arbitrarily mapped to the actual random number streams (actual RNG instances) from omnetpp.ini. The mapping allows for great flexibility in RNG usage and random number streams configuration – even for simulation models which were not written with RNG awareness.

RNG mapping may be specified in omnetpp.ini. The syntax of configuration entries is the following.

```
[General]
<modulepath>.rng-N = M  # where N,M are numeric, M < num-rngs
```

This maps module-local RNG N to physical RNG M. The following example maps all gen module's default (N=0) RNG to physical RNG 1, and all noisychannel module's default (N=0) RNG to physical RNG 2.

```
[General]
num-rngs = 3
**.gen[*].rng-0 = 1
**.noisychannel[*].rng-0 = 2
```

The value also allows expressions, including those containing **index**, **parentIndex**, and **ancestorIndex(level)**. This allows things like assigning a separate RNG to each element of a module vector.

This mapping allows variance reduction techniques to be applied to OMNEST models, without any model change or recompilation.

10.5.4 Automatic Seed Selection

Automatic seed selection is used for an RNG if one does not explicitly specify seeds in omnetpp.ini. Automatic and manual seed selection can co-exist; for a particular simulation, some RNGs can be configured manually, and some automatically.

The automatic seed selection mechanism uses two inputs: the *run number* and the *RNG number*. For the same run number and RNG number, OMNEST always selects the same seed value for any simulation model. If the run number or the RNG number is different, OMNEST does its best to choose different seeds which are also sufficiently separated in the RNG's sequence so that the generated sequences don't overlap.

The run number can be specified either in in omnetpp.ini (e.g. via the **cmdenv-runs-to-execute** option) or on the command line:

```
$ ./mysim -r 1
$ ./mysim -r 2
$ ./mysim -r 3
```

For the `cMersenneTwister` random number generator, selecting seeds so that the generated sequences don't overlap is easy, due to the extremely long sequence of the RNG. The RNG is initialized from the 32-bit seed value $seed = runNumber * numRngs + rngNumber$. (This implies that simulation runs participating in the study should have the same number of RNGs set). [1]

For the `cLCG32` random number generator, the situation is more difficult, because the range of this RNG is rather short ($2^{31}-1$, about 2 billion). For this RNG, OMNEST uses a table of 256 pre-generated seeds, equally spaced in the RNG's sequence. Index into the table is calculated with the $runNumber * numRngs + rngNumber$ formula. Care should be taken that one doesn't exceed 256 with the index, or it will wrap and the same seeds will be used again. It is best not to use the `cLCG32` at all – `cMersenneTwister` is superior in every respect.

10.5.5 Manual Seed Configuration

In some cases, one may want to manually configure seed values. The motivation for doing so may be the use of variance reduction techniques, or the intention to reuse the same seeds for several simulation runs.

To manually set seeds for the Mersenne Twister RNG, use the `seed-`*k*`-mt` option, where *k* is the RNG index. An example:

```
[General]
num-rngs = 3
seed-0-mt = 12
seed-1-mt = 9
seed-2-mt = 7
```

For the now obsolete cLCG32 RNG, the name of the corresponding option is `seed-`*k*`-lcg32`.

10.6 Logging

The OMNEST logging infrastructure provides a few configuration options that affect what is written to the log output. It supports configuring multiple filters: global compile-time, global runtime, and per-component runtime log level filters. For a log statement to actually produce output, it must pass each filter simulatenously. In addition, one can also specify a log prefix format string which determines the context information that is written before each log line. In the following sections, we look how to configure logging.

10.6.1 Compile-Time Filtering

The `COMPILETIME_LOGLEVEL` macro determines which log statements are compiled into the executable. Any log statment which uses a log level below the specified compile-time log level is omitted. In other words, no matter how the runtime log levels are configured, such log

[1] While (to our knowledge) no one has proven that the seeds 0,1,2,... are well apart in the sequence, this is probably true, due to the extremely long sequence of MT. The author would however be interested in papers published about seed selection for MT.

statements are not even executed. This is mainly useful to avoid the performance penalty paid for log statements which are not needed.

```
#define COMPILETIME_LOGLEVEL LOGLEVEL_INFO
EV_INFO << "Packet received successfully" << endl;
EV_DEBUG << "CRC check successful" << endl;
```

In the above example, the output of the second log statement is omitted:

```
[INFO] Packet received successfully
```

If simulation performance is critical, and if there are lots of log statements in the code, it might be useful to omit all log statements from the executable. This can be very simply achieved by putting the following macro into effect for the compilation of all source files.

```
#define COMPILETIME_LOGLEVEL LOGLEVEL_OFF
```

On the other hand, if there's some hard to track down issue, it might be useful to just do the opposite. Compiling with the lowest log level ensures that the log output contains as much information as possible.

```
#define COMPILETIME_LOGLEVEL LOGLEVEL_TRACE
```

By default, the `COMPILETIME_LOGLEVEL` macro is set to `LOGLEVEL_TRACE` if the code is compiled in debug mode (`NDEBUG` is not set). However, it is set to `LOGLEVEL_DETAIL` if the code is compiled in release mode (`NDEBUG` is set).

In fact, the `COMPILETIME_LOG_PREDICATE` macro is the most generic compile time predicate that determines which log statements are compiled into the executable. Mostly, there's no need to redefine this macro, but it can be useful sometimes. For example, one can do compile-time filtering for log categories by redefining this macro. By default, the `COMPILETIME_LOG_PREDICATE` macro is defined as follows:

```
#define COMPILETIME_LOG_PREDICATE(object, loglevel, category) \
        (loglevel >= COMPILETIME_LOGLEVEL)
```

10.6.2 Runtime Filtering

The `cLog::logLevel` variable restricts during runtime which log statements produce output. By default, the global runtime log level doesn't filter logging, it is set to `LOGLEVEL_TRACE`. Although due to its global nature it's not really modular, nevertheless it's still allowed to change the value of this variable. It is mainly used in interactive user interfaces to implement efficient global filtering, but it may also be useful for various debugging purposes.

In addition to the global variable, there's also a per-component runtime log level which only restricts the output of a particular component of the simulation. By default, the runtime log level of all components are set to `LOGLEVEL_TRACE`. Programmatically, these log levels can be retrieved with `cComponent::getLogLevel()` and changed with `cComponent::setLogLevel()`.

In general, any log statment which uses a log level below the specified global runtime log level, or below the specified per-component runtime log level, is omitted. If the log statement appears in a module source, then the module's per-component runtime log level is checked. In any other C++ code, the context module's per-component runtime log level is checked.

In fact, the `cLog::noncomponentLogPredicate` and the `cLog::componentLogPredicate` are the most generic runtime predicates that determines which log statements are executed.

Mostly, there's no need to redefine these predicates, but it can be useful sometimes. For example, one can do runtime filtering for log categories by redefining them. To cite a real example, the `cLog::componentLogPredicate` function contains the following runtime checks:

```
return statementLogLevel >= cLog::loglevel &&
       statementLogLevel >= sourceComponent->getLogLevel() &&
       getEnvir()->isLoggingEnabled(); // for express mode
```

10.6.3 Log Prefix Format

The log prefix format is a string which determines the log prefix that is written before each log line. The format string contains constant parts interleaved with special format directives. The latter always start with the `%` character followed by another character that identifies the format directive. Constant parts are simply written to the output, while format directives are substituted at runtime with the corresponding data that is captured by the log statement.

The following is the list of predefined log prefix format directives. They are organized into groups based on what kind of information they provide.

Log statement related format directives:

- `%l` log level name
- `%c` log category

Current simulation state related format directives:

- `%e` current event number
- `%t` current simulation time
- `%g` current fingerprint if fingerprint verification is enabled in the configuration, otherwise empty
- `%v` current message or event name
- `%a` current message or event class name
- `%n` module name of current event
- `%m` module path of current event
- `%o` module class name of current event
- `%s` simple NED type name of module of current event
- `%q` fully qualified NED type name of module of current event
- `%N` context component name
- `%M` context component path
- `%O` context component class name
- `%S` context component NED type simple name
- `%Q` context component NED type fully qualified name

Simulation run related format directives:

- `%G` config name
- `%R` run number
- `%X` network module class name
- `%Y` network module NED type simple name
- `%Z` network module NED type fully qualified name

C++ source related (where the log statement is) format directives:

- `%p` source object pointer
- `%b` source object name
- `%d` source object path
- `%z` source class name
- `%u` source function name
- `%x` source component NED type simple name
- `%y` source component NED type fully qualified
- `%f` source file name
- `%i` source line number

Operating system related format directives:

- `%w` user time in seconds
- `%W` human readable wall time
- `%H` host name
- `%I` process id

Compound field format directives:

- `%E` event object (class name, name)
- `%U` module of current event (NED type, full path)
- `%C` context component (NED type, full path)
- `%K` context component, if different from current module (NED type, full path)
- `%J` source component or object (NED type or class, full path or pointer)
- `%L` source component or object, if different from context component (NED type or class, full path or pointer)

Padding format directives:

- `%[0-9]+` add spaces until specified column
- `%|` adaptive tabstop: add padding until longest prefix seen so far
- `%>` function call depth times 2-space indentation (see Enter_Method, Enter_Method_Silent)
- `%<` remove preceding whitespace characters

Conditional format directives:

- `%?` ignore the following constant part if the preceding directive didn't print anything (useful for separators)

Escaping the % character:

- `%%` one % character

10.6.4 Configuring Logging in Cmdenv

In Cmdenv, logging can be configured using `omnetpp.ini` configuration options. The configured settings remain in effect during the whole simulation run unless overridden programmatically.

- **`cmdenv-output-file`** redirects standard output to a file
- **`cmdenv-log-prefix`** determines the log prefix of each line
- **`<object-full-path>.cmdenv-log-level`** restricts output on a per-component basis

By default, the log is written to the standard output but it can be redirected to a file. The output can be completely disabled from `omnetpp.ini`, so that it doesn't slow down simulation when it is not needed. The per-component runtime log level option must match the full path of the targeted component. The supported values for this configuration option are the following:

- `off` completely disables log output
- `fatal` omits log output below `LOGLEVEL_FATAL`
- `error` omits log output below `LOGLEVEL_ERROR`
- `warn` omits log output below `LOGLEVEL_WARN`
- `info` omits log output below `LOGLEVEL_INFO`
- `detail` omits log output below `LOGLEVEL_DETAIL`
- `debug` omits log output below `LOGLEVEL_DEBUG`
- `trace` completely enables log output

By default, the log prefix format is set to `"[%l]\t"`. The default setting is intentionally quite simple to avoid cluttered standard output, it produces similar log output:

```
[INFO]  Packet received successfully
[DEBUG] CRC check successful
```

The log messages are aligned vertically because there's a TAB character in the format string. Setting the log prefix format to an empty string disables writing a log prefix altogether. Finally, here is a more detailed format string: "[%l]\t%C for %E: %|", it produces similar output:

```
[INFO]  (IPv4)host.ip for (ICMPMessage)ping0:     Pending (IPv4Datagram)ping0
[INFO]  (ARP)host.arp for (ICMPMessage)ping0:     Starting ARP resolution
[DEBUG] (ARP)host.arp for (ICMPMessage)ping0:     Sending (ARPPacket)arpREQ
[INFO]  (Mac)host.wlan.mac for (ARPPacket)arpREQ: Enqueing (ARPPacket)arpREQ
```

In express mode, for performance reasons, log output is disabled during the whole simulation. However, during the simulation finish stage, logging is automatically re-enabled to allow writing statistical and other results to the log. One can completely disable all logging by adding following configuration option at the beginning of omnetpp.ini:

```
[General]
**.cmdenv-log-level = off
```

Finally, the following is a more complex example that sets the per-component runtime log levels for all PHY components to LOGLEVEL_WARN, except for all MAC modules where it is set to LOGLEVEL_DEBUG, and for all other modules it is set LOGLEVEL_OFF.

```
[General]
**.phy.cmdenv-log-level = warn
**.mac.cmdenv-log-level = debug
**.cmdenv-log-level = off
```

10.6.5 Configuring Logging in Qtenv

The graphical user interface Qtenv provides its own configuration dialog where the user can configure logging. This dialog offers setting the global runtime log level and the log prefix format string. The per-component runtime log levels can be set from the context menu of components. As in Cmdenv, it's also possible to set the log levels to off, effectively disabling logging globally or for specific components only.

In contrast to Cmdenv, setting the runtime log levels is possible even if the simulation is already running. This feature allows continuous control over the level of detail of what is written to the log output. For obvious reasons, changing the log levels has no effect back in time, so already written log content in the log windows will not change.

By default, the log prefix format is set to "%l %C: ", it produces similar log output:

```
INFO  Network.server.wlan[0].mac: Packet received successfully
DEBUG Network.server.wlan[0].mac: CRC check successful
```

Chapter 11

Running Simulations

11.1 Introduction

This chapter describes how to run simulations. It covers basic usage, user interfaces, running simulation campaigns, and many other topics.

11.2 Simulation Executables vs Libraries

As we have seen in the *Build* chapter, simulations may be compiled to an executable or to a shared library. When the build output is an executable, it can be run directly. For example, the Fifo example simulation can be run with the following command:

```
$ ./fifo
```

Simulations compiled to a shared library can be run using the `opp_run` program. For example, if we compiled the Fifo simulation to a shared library on Linux, the build output would be a `libfifo.so` file that could be run with the following command:

```
$ opp_run -l fifo
```

The `-l` option tells `opp_run` to load the given shared library. The `-l` option will be covered in detail in section 11.9.

> **NOTE:** Normal simulation executables like the above `fifo` are also capable of loading additional shared libraries in the same way. What's more, `opp_run` is actually nothing else but a specially-named simulation executable with no model code in it.

11.3 Command-Line Options

The above commands illustrate just the simplest case. Usually you will need to add extra command-line options, for example to specify what ini file(s) to use, which configuration to run, which user interface to activate, where to load NED files from, and so on. The rest of this chapter will cover these options.

To get a complete list of command line options accepted by simulations, run the `opp_run` program (or any other simulation executable) with `-h`:

```
$ opp_run -h
```

Or:

```
$ ./fifo -h
```

11.4 Configuration Options on the Command Line

Configuration options can also be specified on the command line, not only in ini files. To do so, prefix the option name with a double dash, and append the value with an equal sign. Be sure not to have spaces around the equal sign. If the value contains spaces or shell metacharacters, you'll need to protect the value (or the whole option) with quotes or apostrophes.

Example:

```
$ ./fifo --debug-on-errors=true
```

In case an option is specified both on the command line and in an ini file, the command line takes precedence.

To get the list of all possible configuration options, use the `-h config` option. (The additional `-s` option below just makes the output less verbose.)

```
$ opp_run -s -h config
Supported configuration options:
  **.bin-recording=<bool>, default:true; per-object setting
  check-signals=<bool>, default:true; per-run setting
  cmdenv-autoflush=<bool>, default:false; per-run setting
  cmdenv-config-name=<string>; global setting
  ...
```

To see the option descriptions as well, use `-h configdetails`.

```
$ opp_run -h configdetails
```

11.5 Specifying Ini Files

The default ini file is `omnetpp.ini`, and is loaded if no other ini file is given on the command line.

Ini files can be specified both as plain arguments and with the `-f` option, so the following two commands are equivalent:

```
$ ./fifo experiment.ini common.ini
$ ./fifo -f experiment.ini -f common.ini
```

Multiple ini files can be given, and their contents will be merged. This allows for partitioning the configuration into separate files, for example to simulation options, module parameters and result recording options.

11.6 Specifying the NED Path

NED files are loaded from directories listed on the NED path. More precisely, they are loaded from the listed directories and their whole subdirectory trees. Directories are separated with a semicolon (`;`).

> **NOTE:** Semicolon is used as separator on both Unix and Windows.

The NED path can be specified in several ways:

- using the `NEDPATH` environment variable
- using the `-n` command-line option
- in ini files, with the **`ned-path`** configuration option

NED path resolution rules are as follows:

1. OMNEST checks for NED path specified on the command line with the `-n` option
2. If not found on the command line, it checks for the `NEDPATH` environment variable
3. The **`ned-path`** option value from the ini file is appended to the result of the above steps
4. If the result is still empty, it falls back to "." (the current directory)

11.7 Selecting a User Interface

OMNEST simulations can be run under different user interfaces a.k.a. runtime environments. Currently the following user interfaces are supported:

- Qtenv: Qt-based graphical user interface, available since OMNEST 5.0
- Cmdenv: command-line user interface for batch execution

You would typically test and debug your simulation under Qtenv, then run actual simulation experiments from the command line or shell script, using Cmdenv. Qtenv is also better suited for educational and demonstration purposes.

User interfaces are provided in the form of libraries that can be linked with statically, dynamically, or can be loaded at runtime.[1] When several user interface libraries are available in a simulation program, the user can select via command-line or ini file options which one to use. In the absence of such an option, the one with the highest priority will be started. Currently priorities are set such that Qtenv has the highest priority, then Cmdenv. By default, simulations are linked with all available user interfaces, but this can be controlled via `opp_makemake` options or in the OMNEST global build configuration as well. The user interfaces available in a simulation program can be listed by running it the `-h userinterfaces` option.

You can explicitly select a user interface on the command line with the `-u` option (specify `Qtenv` or `Cmdenv` as its argument), or by adding the **`user-interface`** option to the configuration. If both the config option and the command line option are present, the command line option takes precedence.

[1] Via the `-l` option, see section 11.9

Since the graphical interfaces are the default (have higher priority), the most common use of the `-u` option is to select Cmdenv, e.g. for batch execution. The following example performs all runs of the Aloha example simulation using Cmdenv:

```
$ ./aloha -c PureAlohaExperiment -u Cmdenv
```

11.8 Selecting Configurations and Runs

All user interfaces support the `-c <configname>` and `-r <runfilter>` options for selecting which simulation(s) to run.

The `-c` option expects the name of an ini file configuration as an argument. The `-r` option may be needed when the configuration expands to multiple simulation runs. That is the case when the configuration defines a *parameter study* (see section 10.4), or when it contains a **`repeat`** configuration option that prescribes multiple repetitions with different RNG seeds (see section 10.4.6). The `-r` option can then be used to select a subset of all runs (or one specific run, for that matter). A missing `-r` option selects all runs in the given configuration.

It depends on the particular user interface how it interprets the `-c` and `-r` options. Cmdenv performs all selected simulation runs (optionally stopping after the first one that finishes with an error). GUI interfaces like Qtenv may use this information to fill the run selection dialog (or to set up the simulation automatically if there is only one matching run.)

11.8.1 Run Filter Syntax

The run filter accepts two syntaxes: a comma-separated list of run numbers or run number ranges (for example `1,2,5-10`), or an arithmetic expression. The arithmetic expression is similar to constraint expressions in the configuration (see section 10.4.5). It may refer to iteration variables and to the repeat counter with the dollar syntax: `$numHosts`, `$repetition`. An example: `$numHosts>10 && $mean==2`.

Note that due to the presence of the dollar sign (and spaces), the expression should be protected against shell expansion, e.g. using apostrophes:

```
$ ./aloha -c PureAlohaExperiment -r '$numHosts>10 && $mean<2'
```

Or, with double quotes:

```
$ ./aloha -c PureAlohaExperiment -r "\$numHosts>10 && \$mean<2"
```

11.8.2 The Query Option

The `-q` (query) option complements `-c` and `-r`, and allows one to list the runs matched by the run filter. `-q` expects an argument that defines the format and verbosity of the output. Several formats are available: `numruns`, `runnumbers`, `runs`, `rundetails`, `runconfig`. Use `opp_run -h` to get a complete list.

`-q runs` prints one line of information with the iteration variables about each run that the run filter matches. An example:

```
$ ./aloha -s -c PureAlohaExperiment -r '$numHosts>10 && $mean<2' -q runs
Run 14: $numHosts=15, $mean=1, $repetition=0
```

```
Run 15: $numHosts=15, $mean=1, $repetition=1
Run 28: $numHosts=20, $mean=1, $repetition=0
Run 29: $numHosts=20, $mean=1, $repetition=1
```

The `-s` option just makes the output less verbose.

If you need more information, use `-q rundetails` or `-q runconfig`. `rundetails` is like `numruns`, but it also prints the values of the iteration variables and a summary of the configuration (the expanded values of configuration entries that contain iteration variables) for each matching run:

```
$ ./aloha -s -c PureAlohaExperiment -r '$numHosts>10 && $mean<2' -q rundetails
Run 14: $numHosts=15, $mean=1, $repetition=0
    Aloha.numHosts = 15
    Aloha.host[*].iaTime = exponential(1s)

Run 15: $numHosts=15, $mean=1, $repetition=1
    Aloha.numHosts = 15
    Aloha.host[*].iaTime = exponential(1s)
...
```

The `numruns` and `runnumbers` formats are mainly intended for use in scripts. They just print the number of matching runs and the plain run number list, respectively.

```
$ ./aloha -s -c PureAlohaExperiment -r '$numHosts>10 && $mean<2' -q numruns
4
$ ./aloha -s -c PureAlohaExperiment -r '$numHosts>10 && $mean<2' -q runnumbers
 14 15 28 29
```

The `-q` option encapsulates some unrelated functionality, as well: `-q sectioninheritance` ignores `-r`, and prints the inheritance chain of the inifile sections (the inheritance graph after linearization) for the configuration denoted by `-c`.

11.9 Loading Extra Libraries

OMNEST allows you to load shared libraries at runtime. These shared libraries may contain model code (e.g. simple module implementation classes), dynamically registered classes that extend the simulator's functionality (for example NED functions, result filters/recorders, figures types, schedulers, output vector/scalar writers, Qtenv inspectors, or even custom user interfaces), or other code.

> **HINT:** Building shared libraries and loading them dynamically has several advantages over static linking or building executables. Advantages include modularity, reduced build times (versus statically linking a huge executable), and better reuse (being able to use the same library in several projects without change).

Libraries can be specified with the `-l <libraryname>` command line option (there can be several `-l`'s on the command line), or with the **`load-libs`** configuration option. The values from the command line and the config file will be merged.

The prefix and suffix from the library name can be omitted (the extensions `.dll`, `.so`, `.dylib`, and also the common `lib` prefix on Unix systems). This means that you can specify the

library name in a platform independent way: if you specify `-l foo`, then OMNEST will look for `foo.dll`, `libfoo.dll`, `libfoo.so` or `libfoo.dylib`, depending on the platform.

OMNEST will use the `dlopen()` or `LoadLibrary()` system call to load the library. To ensure that the system call finds the file, either specify the library name with a full path (pre- and postfixes of the library file name still can be omitted), or adjust the shared library path environment variable of your OS: `PATH` on Windows, `LD_LIBRARY_PATH` on Unix, and `DYLD_LIBRARY_PATH` on Mac OS X.

> **NOTE:** Runtime loading is not needed if your executable or shared lib was already linked against the library in question. In that case, the platform's dynamic loader will automatically load the library.

11.10 Stopping Condition

The most common way of specifying when to finish the simulation is to set a time limit. There are several time limits that can be set with the following configuration options:

- **`sim-time-limit`** : Limits how long the simulation should run (in simulation time)
- **`cpu-time-limit`** : Limits how much CPU time the simulation can use
- **`real-time-limit`** : Limits how long the simulation can run (in real time)

> **NOTE:** **`cpu-time-limit`** and **`real-time-limit`** may look similar, but in practice, you'll almost always need **`cpu-time-limit`** of the two. Its alternative, **`real-time-limit`** simply measures elapsed time (wall-clock interval), so it does not imply how many cycles the CPU has spent on running your simulation. On a heavily overloaded system where the CPU is shared among a number of computationally intensive jobs, **`real-time-limit`** may stop your simulation much too early.

An example:

```
$ ./fifo --sim-time-limit=500s
```

If several time limits are set together, the simulation will stop when the first one is hit.

If needed, the simulation may also be stopped programmatically, for example when results of a (steady-state) simulation have reached the desired accuracy. This can be done by calling the `endSimulation()` method.

11.11 Controlling the Output

The following options can be used to enable/disable the creation of various output files during simulation.

- **`record-eventlog`** : Turns on the recording of the simulator events into an event log file. The resulting `.elog` file can be analyzed later in the IDE with the Sequence Chart tool.
- **`scalar-recording`** : This option is originally a per-object setting, intended for selectively turning on or off the recording of certain scalar results. However, when it is specified globally to turn off all scalars, no output scalar file (`.sca`) will be created either.

- **vector-recording** : Similar to **scalar-recording**, this option can be used to turn off creating an output vector file (`.vec`).
- **cmdenv-redirect-output** : This is a Cmdenv-specific option, only mentioned here for completeness. It tells Cmdenv to save its standard output to files, one file per run. This option is mainly helpful when running simulation batches.

These configuration options, like any other, can be specified both in ini files and on the command line. An example:

```
$ ./fifo --record-eventlog=true --scalar-recording=false --vector-recording=false
```

11.12 Debugging

Debugging is a task that comes up often during model development. The following configuration options are related to C++ debugging:

- **debug-on-errors** : If the runtime detects any error, it will trigger a debugger trap (programmatic breakpoint) so you will be able to check the location and the context of the problem in your debugger. This option does not start a debugger, the simulation must already have been launched under a debugger.
- **debugger-attach-on-error** : Controls just-in-time debugging. When this option is enabled and an error occurs during simulation, the simulation program will launch an external debugger, and have it attached to the simulation process. Related configuration options are **debugger-attach-on-startup**, **debugger-attach-command** and **debugger-attach-wait-time**.

HINT: Just-in-time debugging is useful when trying to debug a rarely occurring crash in a large simulation batch, or in cases where the simulation is started from a script or another program that cannot be easily modified to start the simulation in a debugger.

An example that launches the simulation under the `gdb` debugger:

```
$ gdb --args ./aloha --debug-on-errors=true
```

11.13 Debugging Leaked Messages

The most common cause of memory leaks in OMNEST simulations is forgetting to delete messages. When this happens, the simulation process will continually grow in size as the simulation progresses, and when left to run long enough, it will eventually cause an out-of-memory condition.

Luckily, this problem is easy to indentify, as all user interfaces display the number of message objects currently in the system. Take a look at the following example Cmdenv output:

```
...
** Event #1908736   t=58914.051870113485   Elapsed: 2.000s (0m 02s)
     Speed:     ev/sec=954368   simsec/sec=29457   ev/simsec=32.3987
```

```
     Messages:  created: 561611   present: 21   in FES: 34
** Event #3433472   t=106067.401570204991   Elapsed: 4.000s (0m 04s)
     Speed:     ev/sec=762368   simsec/sec=23576.7   ev/simsec=32.3357
     Messages:  created: 1010142   present: 354   in FES: 27
** Event #5338880   t=165025.763387178965   Elapsed: 6.000s (0m 06s)
     Speed:     ev/sec=952704   simsec/sec=29479.2   ev/simsec=32.3179
     Messages:  created: 1570675   present: 596   in FES: 21
** Event #6850304   t=211763.433233042017   Elapsed: 8.000s (0m 08s)
     Speed:     ev/sec=755712   simsec/sec=23368.8   ev/simsec=32.3385
     Messages:  created: 2015318   present: 732   in FES: 38
** Event #8753920   t=270587.781554343184   Elapsed: 10.000s (0m 10s)
     Speed:     ev/sec=951808   simsec/sec=29412.2   ev/simsec=32.361
     Messages:  created: 2575634   present: 937   in FES: 32
** Event #10270208   t=317495.244698246477   Elapsed: 12.000s (0m 12s)
     Speed:     ev/sec=758144   simsec/sec=23453.7   ev/simsec=32.3251
     Messages:  created: 3021646   present: 1213   in FES: 20
...
```

The interesting parts are in bold font. The steadily increasing numbers are an indication that the simulation model, i.e. one or more modules in it, are missing some `delete msg` calls. It is best to use Qtenv to narrow down the issue to specific modules and/or message types.

Qtenv is also able to display the number of messages currently in the simulation. The numbers are displayed on the status bar. If you find that the number of messages is steadily increasing, you need to find where the message objects are located. This can be done with the help of the *Find/Inspect Objects* dialog.

If the number of messages is stable, it is still possible that the simulation is leaking other `cObject`-based objects; they can also be found using the *Find/Inspect Objects* dialog.

If the simulation is leaking non-OMNEST objects (i.e. not something derived from `cObject`) or other memory blocks, Cmdenv and Qtenv cannot help in tracking down the issue.

11.14 Debugging Other Memory Problems

Technically, memory leaks are only a subset of problems associated with memory allocations, i.e. the usage of `new` and `delete` in C++.

- *memory leaks*, that is, forgetting to delete objects or memory blocks no longer used, usually just prevents the user from being able to run the simulation program long enough;
- *dereferencing dangling pointers*, i.e. accessing an already deleted object or memory block (or trying to delete one for a second time) usually results in a crash;
- *heap corruption*, caused by e.g. writing past the end of an allocated array, usually also results in a crash.

There are specialized tools that can help in tracking down memory allocation problems (memory leak, double-deletion, referencing deleted blocks, etc). Some of these tools are listed below.

- *Valgrind*, our primary recommendation, is a CPU emulator and memory debugger tool for Linux.

- Other memory debugger libraries/tools include *MemProf*, *MPatrol*, *dmalloc* and *ElectricFence*. Most of these tools support tracking down memory leaks as well as detecting double deletion, writing past the end of an allocated block, etc.
- There are several commercial offerings as well, e.g. *Purify* and *Insure++*.

11.15 Profiling

When a simulation runs correctly but is too slow, you might want to *profile* it. Profiling basically means collecting runtime information about how much time is spent at various parts of the program, in order to find places where optimizing the code would have the most impact.

However, there are a few other options you can try before resorting to profiling and optimizing. First, verify that it is the simulation itself that is slow. Make sure features like eventlog recording is not accidentally turned on. Run the simulation under Cmdenv to eliminate any possible overhead from Qtenv. If you must run the simulation under Qtenv, you can still gain speed by disabling animation features, closing all inspectors, hiding UI elements like the timeline, and so on.

Also, compile your code in release mode (with `make MODE=release`, see 9.2.3) instead of debug. That can make a huge difference, especially with heavily templated code.

> **HINT:** If you decide to optimize the program, we recommend that you don't skip the profiling step. Even for experienced programmers, a profiling session is often full of surprises, and CPU time is spent at other places than one would expect.

Some profiling software:

- **Debuggers**. A very simple but frequently useful way of profiling is stopping the program in a debugger from time to time, and looking at the stack trace before resuming (manual statistical profiling). If the program always stops at the same place in the code, that might be the bottleneck.
- **Valgrind/KCachegrind**. KCachegrind is a graphical visualizer for traces generated by *valgrind* and its *callgrind* tool in Linux. These tools are free and open source software, packaged with most Linux distributions.
- There are also commercial C/C++ profilers like RotateRight's Zoom. Profilers are also part of larger packages like PurifyPlus or Parasoft Insure++.

11.16 Checkpointing

Debugging long-running simulations can be challenging, because one often needs to run the simulation for a long time just to get to the point of failure and be able to start debugging.

Checkpointing can facilitate debugging such errors. It is a technique that basically consists of saving a snapshot of the application's state, and being able to resume execution from there, even multiple times. OMNEST itself contains no checkpointing functionality, but it is available via external tools. It depends on the tool whether it is able to restore GUI windows (usually not.)

Some checkpointing software that is available on Linux:

- Berkeley Lab Checkpoint/Restart (BLCR)
- DMTCP (Distributed MultiThreaded Checkpointing)
- CRIU is a user space checkpoint lib
- Docker and the underlying technology contain a checkpoint and restore mechanism

11.17 Using Cmdenv

Cmdenv is a lightweight, command line user interface that compiles and runs on all platforms. Cmdenv is designed primarily for batch execution.

Cmdenv simply executes some or all simulation runs that are described in the configuration file. The runs to be executed can be passed via command-line arguments or configuration options.

Cmdenv runs simulations in the same process. This means that e.g. if one simulation run writes a global variable, subsequent runs will also see the change. This is one reason why global variables in models are strongly discouraged.

11.17.1 Sample Output

When you run the Fifo example under Cmdenv, you should see something like this:

```
$ ./fifo -u Cmdenv -c Fifo1

OMNeT++ Discrete Event Simulation  (C) 1992-2017 Andras Varga, OpenSim Ltd.
Version: 5.0, edition: Academic Public License -- NOT FOR COMMERCIAL USE
See the license for distribution terms and warranty disclaimer
Setting up Cmdenv...
Loading NED files from .: 5

Preparing for running configuration Fifo1, run #0...
Scenario: $repetition=0
Assigned runID=Fifo1-0-20090104-12:23:25-5792
Setting up network 'FifoNet'...
Initializing...
Initializing module FifoNet, stage 0
Initializing module FifoNet.gen, stage 0
Initializing module FifoNet.fifo, stage 0
Initializing module FifoNet.sink, stage 0

Running simulation...
** Event #1   t=0   Elapsed: 0.000s (0m 00s)  0% completed
     Speed:     ev/sec=0   simsec/sec=0   ev/simsec=0
     Messages:  created: 2   present: 2   in FES: 1
** Event #232448   t=11719.051014922336   Elapsed: 2.003s (0m 02s)  3% completed
     Speed:     ev/sec=116050   simsec/sec=5850.75   ev/simsec=19.8351
     Messages:  created: 58114   present: 3   in FES: 2
```

```
...
** Event #7206882   t=360000.52066583684   Elapsed: 78.282s (1m 18s)  100% complet
     Speed:     ev/sec=118860   simsec/sec=5911.9   ev/simsec=20.1053
     Messages:  created: 1801723   present: 3   in FES: 2

<!> Simulation time limit reached -- simulation stopped.

Calling finish() at end of Run #0...
End.
```

As Cmdenv runs the simulation, it periodically prints the sequence number of the current event, the simulation time, the elapsed (real) time, and the performance of the simulation (how many events are processed per second; the first two values are 0 because there wasn't enough data for it to calculate yet). At the end of the simulation, the `finish()` methods of the simple modules are run, and the outputs from them are displayed.

11.17.2 Selecting Runs, Batch Operation

The most important command-line options for Cmdenv are `-c` and `-r` for selecting which simulations to perform. (They were described in section 11.8.) They also have their equivalent configuration options that can be written in files as well: **cmdenv-config-name** and **cmdenv-runs-to-execute**.

Another configuration option, **cmdenv-stop-batch-on-error** controls Cmdenv's behavior when performing multiple runs: it determines whether Cmdenv should stop after the first run that finishes with an error. By default, it does.

When performing multiple runs, Cmdenv prints run statistics at the end. Example output:

```
$ ./aloha -c PureAlohaExperiment -u Cmdenv
...
Run statistics: total 42, successful 30, errors 1, skipped 11
```

11.17.3 Express Mode

Cmdenv can execute simulations in two modes:

- **Normal** (non-express) mode is for debugging; detailed information will be written to the standard output (event banners, module log, etc).
- **Express** mode can be used for long simulation runs; only periodical status updates are displayed about the progress of the simulation.

The default mode is Express. To turn off Express mode, specify `false` for the **cmdenv-express-mode** configuration option:

```
$ ./fifo -u Cmdenv -c Fifo1 --cmdenv-express-mode=false
```

There are several other options that also affect Express-mode and Normal mode behavior:

- Express: **cmdenv-performance-display**, **cmdenv-status-frequency**

- Normal: **cmdenv-event-banners**, **cmdenv-event-banner-details**, **cmdenv-log-level**, **cmdenv-log-prefix**, etc.

See Appendix I for more information about these options.

Interpreting Express-Mode Output

When the simulation is running in Express mode with detailed performance display enabled (`cmdenv-performance-display=true`), Cmdenv periodically outputs a three-line status report about the progress of the simulation. The output looks like this:

```
...
** Event #250000   t=123.74354 ( 2m  3s)    Elapsed: 0m 12s
     Speed:     ev/sec=19731.6   simsec/sec=9.80713   ev/simsec=2011.97
     Messages:  created: 55532   present: 6553   in FES: 8
** Event #300000   t=148.55496 ( 2m 28s)    Elapsed: 0m 15s
     Speed:     ev/sec=19584.8   simsec/sec=9.64698   ev/simsec=2030.15
     Messages:  created: 66605   present: 7815   in FES: 7
...
```

The first line of the status display (beginning with `**`) contains:

- how many events have been processed so far
- the current simulation time (t), and
- the elapsed time (wall clock time) since the beginning of the simulation run.

The second line displays simulation performance metrics:

- `ev/sec` indicates *performance*: how many events are processed in one real-time second. On one hand it depends on your hardware (faster CPUs process more events per second), and on the other hand it depends on the complexity (amount of calculations) associated with processing one event. For example, protocol simulations tend to require more processing per event than e.g. queueing networks, thus the latter produce higher ev/sec values. In any case, this value is largely independent of the size of your model, i.e. the number of modules in it.
- `simsec/sec` shows *relative speed* of the simulation, that is, how fast the simulation is progressing compared to real time, how many simulated seconds can be done in one real second. This value virtually depends on everything: on the hardware, on the size of the simulation model, on the complexity of events, and the average simulation time between events as well.
- `ev/simsec` is the *event density*: how many events are there per simulated second. Event density only depends on the simulation model, regardless of the hardware used to simulate it: in a high-speed optical network simulation you will have very high values (10^9), while in a call center simulation this value is probably well under 1. It also depends on the size of your model: if you double the number of modules in your model, you can expect the event density to double, too.

The third line displays the number of messages, and it is important because it may indicate the "health" of your simulation.

- `Created`: total number of message objects created since the beginning of the simulation run. This does not mean that this many message object actually exist, because some (many) of them may have been deleted since then. It also does not mean that *you* created all those messages – the simulation kernel also creates messages for its own use (e.g. to implement `wait()` in an `activity()` simple module).
- `Present`: the number of message objects currently present in the simulation model, that is, the number of messages created (see above) minus the number of messages already deleted. This number includes the messages in the FES.
- `In FES`: the number of messages currently scheduled in the Future Event Set.

The second value, the number of messages present, is more useful than perhaps one would initially think. It can be an indicator of the "health" of the simulation; if it is growing steadily, then either you have a memory leak and losing messages (which indicates a programming error), or the network you simulate is overloaded and queues are steadily filling up (which might indicate wrong input parameters).

Of course, if the number of messages does not increase, it does not mean that you do *not* have a memory leak (other memory leaks are also possible). Nevertheless the value is still useful, because by far the most common way of leaking memory in a simulation is by not deleting messages.

11.17.4 Other Options

Cmdenv has more configuration options than mentioned in this section; see the options beginning with `cmdenv-` in Appendix I for the complete list.

11.18 The Qtenv Graphical User Interface

Qtenv is a runtime simulation GUI. Qtenv supports interactive simulation execution, animation, tracing and debugging. Qtenv is recommended in the development stage of a simulation and for presentation purposes, since it allows one to get a detailed picture of the state of simulation at any point of execution and to follow what happens inside the network. Note that 3D visualization support and smooth animation support are only available in Qtenv.

> **NOTE:** This section only covers the command-line and configuration options of Qtenv; the user interface is described in the Qtenv chapter of the OMNEST User Guide.

11.18.1 Command-Line and Configuration Options

Simulations run under Qtenv accept all general command line and configuration options, including `-c` and `-r`. The configuration options specific to Qtenv include:

- **`qtenv-default-config`**: Specifies which config Qtenv should set up automatically on startup. The default is to ask the user. This option is equivalent to the `-c` command-line option.
- **`qtenv-default-run`**: Specifies which run (of the default config, see qtenv-default-config) Qtenv should set up automatically on startup. The default is to ask the user. This option is equivalent to the `-r` command-line option.

- **qtenv-extra-stack**: Specifies the extra amount of stack that is reserved for each activity() simple module when the simulation is run under Qtenv.

Qtenv is also affected by the following option:

- **image-path**: Specifies the path for loading module icons.

See Appendix I for the list of possible configuration options.

11.19 Running Simulation Campaigns

Once your model works reliably, you will usually want to run several simulations, either to explore the parameter space via a *parameter study* (see section 10.4), or to do multiple repetitions with different RNG seeds to increase the statistical accuracy of the results (see section 10.4.6).

In this section, we will explore several ways to run batches of simulations efficently.

11.19.1 The Naive Approach

Assume that you want to run the parameter study in the Aloha example simulation for the $numHosts > 15$ cases.

The first idea is that Cmdenv is capable of running simulation batches. The following command will do the job:

```
$ ./aloha -u Cmdenv -c PureAlohaExperiment -r '$numHosts>15'
...
Run statistics: total 14, successful 14
End.
```

It works fine. However, this approach has some drawbacks which becomes apparent when running hundreds or thousands of simulation runs.

1. It uses only one CPU. In the age of multi-core CPUs, this is not very efficient.
2. More prone to C++ programming errors in the model. A failure in a single run may abort execution (segfault) or corrupt the process state, possibly invalidating the results of subsequent runs.

To address the second drawback, we can execute each simulation run in its own Cmdenv instance.

```
$ ./aloha -c PureAlohaExperiment -r '$numHosts>15' -s -q runnumbers
28 29 30 31 32 33 34 35 36 37 38 39 40 41
$ ./aloha -u Cmdenv -c PureAlohaExperiment -r 28
$ ./aloha -u Cmdenv -c PureAlohaExperiment -r 29
$ ./aloha -u Cmdenv -c PureAlohaExperiment -r 30
...
$ ./aloha -u Cmdenv -c PureAlohaExperiment -r 41
```

It's a lot of commands to issue manually, but luckily it can be automated with a shell script like this:

```
#! /bin/sh
RUNS=$(./aloha -c PureAlohaExperiment -r '$numHosts>15' -s -q runnumbers)
for i in $RUNS; do
    ./aloha -u Cmdenv -c PureAlohaExperiment -r $i
done
```

Save the above into a text file called e.g. `runAloha`. Then give it executable permission, and run it:

```
$ chmod +x runAloha
$ ./runAloha
```

It will execute the simulations one-by-one, each in its own Cmdenv instance.

This approach involves a process start overhead for each simulation. Normally, this overhead is small compared to the time spent simulating. However, it may become more of a problem when running a large number of very short simulations («1s in CPU time). This effect may be mitigated by letting Cmdenv do several (e.g. 10) simulations in one go.

And then, the script still uses only one CPU. It would be better to keep all CPUs busy. For example, if you have 8 CPUs, there should be eight processes running all the time – when one terminates, another would be launched in its place. You might notice that this behavior is similar to what GNU Make's `-j<numJobs>` option does. The `opp_runall` utility, to be covered in the next section, exploits GNU Make to schedule the running of simulations on multiple CPUs.

11.19.2 Using opp_runall

OMNEST has a utility program called `opp_runall`, which allows you to execute simulations using multiple CPUs and multiple processes.

`opp_runall` groups simulation runs into batches. Every batch corresponds to a Cmdenv process, that is, runs of a batch execute sequentially inside the same Cmdenv process. Batches (i.e. Cmdenv instances) are scheduled for running so that they keep all CPUs busy. The batch size as well as the number of CPUs to use have sensible defaults but can be overridden.

Command Line

`opp_runall` expects the normal simulation command in its argument list. The first positional (non-option) argument and all following arguments are treated as the simulation command (simulation program and its arguments).

Thus, to modify a normal Cmdenv simulation command to make use of multiple CPUs, simply prefix it with `opp_runall`:

```
$ opp_runall ./aloha -u Cmdenv -c PureAlohaExperiment -r '$numHosts>15'
```

Options intended for `opp_runall` should come before the the simulation command. These options include `-b<N>` for specifying the batch size, and `-j<N>` to specify the number of CPUs to use.

```
$ opp_runall -j8 -b4 ./aloha -u Cmdenv -c PureAlohaExperiment -r '$numHosts>15'
```

How It Works

First, `opp_runall` invokes the simulation command with extra command arguments (`-s -q runnumbers`) to figure out the list of runs it needs to perform, and groups the run numbers into batches. Then it exploits GNU make and its `-j<N>` option to do the heavy lifting. Namely, it generates a temporary makefile that allows `make` to run batches in parallel, and invokes `make` with the appropriate `-j` option. It is also possible to export the makefile for inspection and/or running it manually.

To illustrate the above, here is the content of such a makefile:

```
#
# This makefile was generated with the following command:
# opp_runall -j2 -b4 -e tmp ./aloha -u Cmdenv -c PureAlohaExperiment -r $numHosts>
#

SIMULATIONCMD = ./aloha -u Cmdenv -c PureAlohaExperiment -s \
                --cmdenv-redirect-output=true
TARGETS =  batch0 batch1 batch2 batch3

.PHONY: $(TARGETS)

all: $(TARGETS)
    @echo All runs completed.

batch0:
    $(SIMULATIONCMD) -r 28,29,30,31

batch1:
    $(SIMULATIONCMD) -r 32,33,34,35

batch2:
    $(SIMULATIONCMD) -r 36,37,38,39

batch3:
    $(SIMULATIONCMD) -r 40,41
```

11.19.3 Exploiting Clusters

With large scale simulations, using one's own desktop computer might not be enough. The solution could be to run the simulation on remote machines, that is, to employ a computing cluster.

In simple setups, cross-mounting the file system that contains OMNEST and the model, and using `ssh` to run the simulations might already provide a good solution.

In other cases, submitting simulation jobs and harvesting the results might be done via batch-queuing, cluster computing or grid computing middleware. The following list contains some pointers to such software:

- **HTCondor**, previously called **Condor**, is an open source software package that enables High Throughput Computing (HTC) on large collections of distributively owned computing resources. HTCondor can manage a dedicated cluster of workstations, and it can

also harness non-dedicated, preexisting resources under distributed ownership. A user can submit jobs to HTCondor. HTCondor finds an available machine on the network and begins running the job on that machine. HTCondor also supports checkpointing and migrating jobs.

- **Open Grid Scheduler/Grid Engine** is a commercially supported open-source batch-queuing system for distributed resource management. OGS/GE is based on Sun Grid Engine (SGE), and maintained by the same group of external (i.e. non-Sun) developers who started contributing code since 2001. There is also a commercial SGE successor, **Univa Grid Engine**, formerly called Oracle Grid Engine.

- **Slurm Workload Manager**, or Slurm, is a free and open-source job scheduler for Linux and Unix-like kernels, used by many of the world's supercomputers and computer clusters.

- **Apple's Xgrid** has unfortunately been removed from Mac OS X with the release of Mountain Lion (2012). Xgrid was distributed computing for the masses – easy, plug and play, not complicated. You could network your Mac computers together, and use that power on one computer to do something that took a lot of computing power. Currently, Pooch is advertised as software providing the easiest way to assemble and operate a high-performance parallel computer from Macs.

11.20 Akaroa Support: Multiple Replications in Parallel

11.20.1 Introduction

Typical simulations are Monte-Carlo simulations: they use (pseudo-)random numbers to drive the simulation model. For the simulation to produce statistically reliable results, one has to carefully consider the following:

- When the initial transient is over, when can we start collecting data? We usually don't want to include the initial transient when the simulation is still "warming up."

- When can we stop the simulation? We want to wait long enough so that the statistics we are collecting can "stabilize", or reach the required sample size to be statistically trustable.

Neither question is trivial to answer. One might just suggest to wait "very long" or "long enough". However, this is neither simple (how do you know what is "long enough"?) nor practical (even with today's high speed processors simulations of modest complexity can take hours, and one may not afford multiplying runtimes by, say, 10, "just to be safe.") If you need further convincing, please read [PJL02] and be horrified.

A possible solution is to look at the statistics while the simulation is running, and decide at runtime when enough data have been collected for the results to have reached the required accuracy. One possible criterion is given by the confidence level, more precisely, by its width relative to the mean. But ex ante it is unknown how many observations have to be collected to achieve this level – it must be determined at runtime.

11.20.2 What Is Akaroa

Akaroa [EPM99] addresses the above problem. According to its authors, Akaroa (Akaroa2) is a "fully automated simulation tool designed for running distributed stochastic simulations in MRIP scenario" in a cluster computing environment.

MRIP stands for *Multiple Replications in Parallel*. In MRIP, the computers of the cluster run independent replications of the whole simulation process (i.e. with the same parameters but different seed for the RNGs (random number generators)), generating statistically equivalent streams of simulation output data. These data streams are fed to a global data analyser responsible for analysis of the final results and for stopping the simulation when the results reach a satisfactory accuracy.

The independent simulation processes run independently of one another and continuously send their observations to the central analyser and control process. This process *combines* the independent data streams, and calculates from these observations an overall estimate of the mean value of each parameter. Akaroa2 decides by a given confidence level and precision whether it has enough observations or not. When it judges that is has enough observations it halts the simulation.

If n processors are used, the needed simulation execution time is usually n times smaller compared to a one-processor simulation (the required number of observations are produced sooner). Thus, the simulation would be sped up approximately in proportion to the number of processors used and sometimes even more.

Akaroa was designed at the University of Canterbury in Christchurch, New Zealand and can be used free of charge for teaching and non-profit research activities.

11.20.3 Using Akaroa with OMNEST

Starting Akaroa

Before the simulation can be run in parallel under Akaroa, you have to start up the system:

- Start `akmaster` running in the background on some host.
- On each host where you want to run a simulation engine, start `akslave` in the background.

Each `akslave` establishes a connection with the `akmaster`.

Then you use `akrun` to start a simulation. `akrun` waits for the simulation to complete, and writes a report of the results to the standard output. The basic usage of the `akrun` command is:

```
$ akrun -n num_hosts command [argument..]
```

where *command* is the name of the simulation you want to start. Parameters for Akaroa are read from the file named `Akaroa` in the working directory. Collected data from the processes are sent to the `akmaster` process, and when the required precision has been reached, `akmaster` tells the simulation processes to terminate. The results are written to the standard output.

The above description is not detailed enough to help you set up and successfully use Akaroa – for that you need to read the Akaroa manual.

Configuring OMNEST for Akaroa

First of all, you have to compile OMNEST with Akaroa support enabled.

The OMNEST simulation must be configured in `omnetpp.ini` so that it passes the observations to Akaroa. The simulation model itself does not need to be changed – it continues to write the observations into output vectors (`cOutVector` objects, see chapter 7). You can place some of the output vectors under Akaroa control.

You need to add the following to `omnetpp.ini`:

```
[General]
rng-class = "cAkaroaRNG"
outputvectormanager-class = "cAkOutputVectorManager"
```

These lines cause the simulation to obtain random numbers from Akaroa, and allows data written to selected output vectors to be passed to Akaroa's global data analyser. [2]

Akaroa's RNG is a Combined Multiple Recursive pseudorandom number generator (CMRG) with a period of approximately 2^{191} random numbers, and provides a unique stream of random numbers for every simulation engine.

> **NOTE:** It is vital that you obtain random numbers from Akaroa; otherwise, all simulation processes will run with the same RNG seeds, and produce exactly the same results.

Then you need to specify which output vectors you want to be under Akaroa control (by default, none of them are). You can use the `*`, `**` wildcards (see section 10.3.1) to place certain vectors under Akaroa control.

```
<modulename>.<vectorname1>.with-akaroa = true
<modulename>.<vectorname2>.with-akaroa = true
```

Using Shared File Systems

It is usually practical to have the same physical disk mounted (e.g. via NFS or Samba) on all computers in the cluster. However, because all OMNEST simulation processes run with the same settings, they would overwrite each other's output files. Your can prevent this from happening using the **`fname-append-host`** ini file entry:

```
[General]
fname-append-host = true
```

When turned on, it appends the host name to the names of the output files (output vector, output scalar, snapshot files).

[2] For more details on the plugin mechanism these settings make use of, see 17.

Chapter 12

Result Recording and Analysis

12.1 Result Recording

OMNEST provides built-in support for recording simulation results, via *output vectors* and *output scalars*. Output vectors are time series data, recorded from simple modules or channels. You can use output vectors to record end-to-end delays or round trip times of packets, queue lengths, queueing times, module state, link utilization, packet drops, etc. – anything that is useful to get a full picture of what happened in the model during the simulation run.

Output scalars are summary results, computed during the simulation and written out when the simulation completes. A scalar result may be an (integer or real) number, or may be a statistical summary comprised of several fields such as count, mean, standard deviation, sum, minimum, maximum, etc., and optionally histogram data.

Results may be collected and recorded in two ways:

1. Based on the signal mechanism, using declared statistics;
2. Directly from C++ code, using the simulation library

The second method has been the traditional way of recording results. The first method, based on signals and declared statistics, was introduced in OMNEST 4.1, and it is preferable because it allows you to always record the results in the form you need, without requiring heavy instrumentation or continuous tweaking of the simulation model.

12.1.1 Using Signals and Declared Statistics

This approach combines the signal mechanism (see 4.14) and NED properties (see 3.12) in order to de-couple the generation of results from their recording, thereby providing more flexibility in what to record and in which form. The details of the solution have been described in section 4.15 in detail; here we just give a short overview.

Statistics are declared in the NED files with the `@statistic` property, and modules emit values using the signal mechanism. The simulation framework records data by adding special result file writer listeners to the signals. By being able to choose what listeners to add, the user can control what to record in the result files and what computations to apply before recording. The aforementioned section 4.15 also explains how to instrument simple modules and channels for signals-based result recording.

The signals approach allows for calculation of aggregate statistics (such as the total number of packet drops in the network) and for implementing a warm-up period without support from module code. It also allows you to write dedicated statistics collection modules for the simulation, also without touching existing modules.

The same configuration options that were used to control result recording with `cOutVector` and `recordScalar()` also work when utilizing the signals approach, and there are extra configuration options to make the additional possibilities accessible.

12.1.2 Direct Result Recording

With this approach, scalar and statistics results are collected in class variables inside modules, then recorded in the finalization phase via `recordScalar()` calls. Vectors are recorded using `cOutVector` objects. Use `cStdDev` to record summary statistics like mean, standard deviation, minimum/maximum, and histogram-like classes (`cHistogram`, `cPSquare`, `cKSplit`) to record the distribution. These classes are described in sections 7.9 and 7.10. Recording of individual vectors, scalars and statistics can be enabled or disabled via the configuration (ini file), and it is also the place to set up recording intervals for vectors.

The drawback of recording results directly from modules is that result recording is hardcoded in modules, and even simple requirement changes (e.g. record the average delay instead of each delay value, or vice versa) requires either code change or an excessive amount of result collection code in the modules.

12.2 Configuring Result Collection

12.2.1 Result File Names

Simulation results are recorded into *output scalar files* that actually hold statistics results as well, and *output vector files*. The usual file extension for scalar files is `.sca`, and for vector files `.vec`.

Every simulation run generates a single scalar file and a vector file. The file names can be controlled with the **`output-vector-file`** and **`output-scalar-file`** options. These options rarely need to be used, because the default values are usually fine. The defaults are:

```
output-vector-file = "${resultdir}/${configname}-${runnumber}.vec"
output-scalar-file = "${resultdir}/${configname}-${runnumber}.sca"
```

Here, `${resultdir}` is the value of the **`result-dir`** configuration option which defaults to `results/`, and `${configname}` and `${runnumber}` are the name of the configuration name in the ini file (e.g. `[Config PureAloha]`), and the run number. Thus, the above defaults generate file names like `results/PureAloha-0.vec`, `results/PureAloha-1.vec`, and so on.

NOTE: In OMNEST 3.x, the default result file names were `omnetpp.vec` and `omnetpp.sca`, and scalar files were always appended to, rather than being overwritten as in the 4.x version. When needed, the old behavior for scalar files can be turned back on by setting `output-scalar-file-append=true` in the configuration.

12.2.2 Enabling/Disabling Result Items

The recording of simulation results can be enabled/disabled at multiple levels with various configuration options:

- All recording from a `@statistic` can be enabled/disabled together using the **`statistic-recording`** option;
- Recording of a scalar or a statistic object can be controlled with the **`scalar-recording`** option;
- Recording of an output vector can be controlled with the **`vector-recording`** option;
- Recording of the bins of a histogram object can be controlled with the **`bin-recording`** option.

All the above options are boolean per-object options, thus, they have similar syntaxes:

- *<module-path>.<statistic-name>*.**`statistic-recording`** `= true/false`
- *<module-path>.<scalar-name>*.**`scalar-recording`** `= true/false`
- *<module-path>.<vector-name>*.**`vector-recording`** `= true/false`
- *<module-path>.<histogram-name>*.**`bin-recording`** `= true/false`

For example, all recording from the following statistic

```
@statistic[queueLength](record=max,timeavg,vector);
```

can disabled with this ini file line:

```
**.queueLength.statistic-recording = false
```

When a scalar, vector, or histogram is recorded using a `@statistic`, its name is derived from the statistic name, by appending the recording mode after a semicolon. For example, the above statistic will generate the scalars named `queueLength:max` and `queueLength:timeavg`, and the vector named `queueLength:vector`. Their recording can be individually disabled with the following lines:

```
**.queueLength:max.scalar-recording = false
**.queueLength:timeavg.scalar-recording = false
**.queueLength:vector.vector-recording = false
```

The statistic, scalar or vector name part in the key may also contain wildcards. This can be used, for example, to handle result items with similar names together, or, by using `*` as name, for filtering by module or to disable all recording. The following example turns off recording of all scalar results except those called `latency`, and those produced by modules named `tcp`:

```
**.tcp.*.scalar-recording = true
**.latency.scalar-recording = true
**.scalar-recording = false
```

To disable all result recording, use the following three lines:

```
**.statistic-recording = false
**.scalar-recording = false
**.vector-recording = false
```

The first line is not strictly necessary. However, it may improve runtime performance because it causes result recorders not to be added, instead of adding and then disabling them.

12.2.3 Selecting Recording Modes for Signal-Based Statistics

Signal-based statistics recording has been designed so that it can be easily configured to record a "default minimal" set of results, a "detailed" set of results, and a custom set of results (by modifying the previous ones, or defined from scratch).

Recording can be tuned with the **`result-recording-modes`** per-object configuration option. The "object" here is the statistic, which is identified by the full path (hierarchical name) of the module or connection channel object in question, plus the name of the statistic (which is the "index" of `@statistic` property, i.e. the name in the square brackets). Thus, configuration keys have the syntax *<module-path>.<statistic-name>*`.result-recording-modes=`.

The **`result-recording-modes`** option accepts one or more items as value, separated by comma. An item may be a result recording mode (surprise!), and two words with a special meaning, `default` and `all`:

- A *result recording mode* means any item that may occur in the `record` key of the `@statistic` property; for example, `count`, `sum`, `mean`, `vector((count-1)/2)`.
- **`default`** stands for the set of non-optional items from the `@statistic` property's `record` list, that is, those without question marks.
- **`all`** means all items from the `@statistic` property's `record` list, including the ones with question marks.

The default value is `default`.

A lone "-" as option value disables all recording modes.

Recording mode items in the list may be prefixed with "+" or "-" to add/remove them from the set of result recording modes. The initial set of result recording modes is `default`; if the first item is prefixed with "+" or "-", then that and all subsequent items are understood as modifying the set; if the first item does not start with with "+" or "-", then it replaces the set, and further items are understood as modifying the set.

This sounds more complicated than it is; an example will make it clear. Suppose we are configuring the following statistic:

```
@statistic[foo](record=count,mean,max?,vector?);
```

With the following the ini file lines (see results in comments):

```
**.result-recording-modes = default   # --> count, mean
**.result-recording-modes = all       # --> count, mean, max, vector
**.result-recording-modes = -         # --> none
**.result-recording-modes = mean      # --> only mean (disables 'default')
**.result-recording-modes = default,-vector,+histogram # --> count,mean,histogram
**.result-recording-modes = -vector,+histogram         # --> same as the previous
**.result-recording-modes = all,-vector,+histogram  # --> count,mean,max,histogram
```

Here is another example which shows how to write a more specific option key. The following line applies to `queueLength` statistics of `fifo[]` submodule vectors anywhere in the network:

```
**.fifo[*].queueLength.result-recording-modes = +vector  # default plus vector
```

In the result file, the recorded scalars will be suffixed with the recording mode, i.e. the mean of `queueingTime` will be recorded as `queueingTime:mean`.

12.2.4 Warm-up Period

The **`warmup-period`** option specifies the length of the initial warm-up period. When set, results belonging to the first x seconds of the simulation will not be recorded into output vectors, and will not be counted into the calculation of output scalars. This option is useful for steady-state simulations. The default is 0s (no warmup period).

Example:

```
warmup-period = 20s
```

Refining Warm-up Period Handling

Warm-up period handling works by inserting a special filter, a *warm-up period filter* into the filter/recorder chain if a warm-up period is requested. This filter acts like a timed switch: it discards values during the specified warm-up period, and allows them to pass through afterwards.

OMNEST allows you to disable the automatic adding of warm-up filters by specifying `auto-WarmupFilter=false` in the `@statistic` as an attribute, and manually placing such filters (`warmup`) instead.

Why is this necessary? By default, the filter is inserted at the front of the filter/recorder chain of every statistic. However, the front is not always the correct place for the warm-up period filter. Consider for example, computing the number of packets in a (compound) queue as the difference between the number of arrivals and departures from the queue. This can be achieved using `@statistic` as follows:

```
@signal[pkIn](type=cPacket);
@signal[pkOut](type=cPacket);
@statistic[queueLen](source=count(pkIn)-count(pkOut);record=vector);
```

When a warm-up period is configured, the necessary warm-up period filters are inserted right before the *count* filters. This can be expressed as the following expression for the statistic's `source` attribute:

```
count(warmup(pkIn)) - count(warmup(pkOut))
```

which is apparently incorrect, because the *count* filters only start counting when the warm-up period is over. Thus, the measured queue length will start from zero when the warm-up period is over, even though the queue might not be empty! In fact, if the first event after the warm-up period is a departure, the measured queue length will even go negative.

The right solution would be put the `warmup` filter at the end, like so:

```
warmup(count(pkIn)-count(pkOut))
```

Thus, the correct form of the queue length statistic is the following:

```
@statistic[queueLen](source=warmup(count(pkIn)-count(pkOut));
                     autoWarmupFilter=false;
                     record=vector);
```

Manual Result Recording

Results recorded via signal-based statistics automatically obey the warm-up period setting, but modules that compute and record scalar results manually (via `recordScalar()`) need to be modified so that they take the warm-up period into account.

> **NOTE:** When configuring a warm-up period, make sure that modules that compute and record scalar results manually via `recordScalar()` actually obey the warm-up period in the C++ code.

The warm-up period is available via the `getWarmupPeriod()` method of the simulation manager object, so the C++ code that updates the corresponding state variables needs to be surrounded with an *if* statement:

Old:

```
dropCount++;
```

New:

```
if (simTime() >= getSimulation()->getWarmupPeriod())
    dropCount++;
```

12.2.5 Output Vectors Recording Intervals

The size of output vector files can easily reach several gigabytes, but very often, only some of the recorded statistics are interesting to the analyst. In addition to selecting which vectors to record, OMNEST also allows one to specify one or more collection intervals.

The latter can be configured with the **`vector-recording-intervals`** per-object option. The syntax of the configuration option is *<module-path>.<vector-name>*`.vector-recording-intervals=`*<inte*
where both *<module-path>* and *<vector-name>* may contain wildcards (see 10.3.1). *<vector-name>* is the vector name, or the name string of the `cOutVector` object. By default, all output vectors are turned on for the whole duration the simulation.

One can specify one or more intervals in the *<startTime>..<stopTime>* syntax, separated by comma. *<startTime>* or *<stopTime>* need to be given with measurement units, and both can be omitted to denote the beginning and the end of the simulation, respectively.

The following example limits all vectors to three intervals, except `dropCount` vectors which will be recorded during the whole simulation run:

```
**.dropCount.vector-recording-intervals = 0..
**.vector-recording-intervals = 0..1000s, 5000s..6000s, 9000s..
```

12.2.6 Recording Event Numbers in Output Vectors

A third per-vector configuration option is **`vector-record-eventnumbers`**, which specifies whether to record event numbers for an output vector. (Simulation time and value are always

recorded. Event numbers are needed by the Sequence Chart Tool, for example.) Event number recording is enabled by default; it may be turned off to save disk space.

```
**.vector-record-eventnumbers = false
```

If the (default) `cIndexedFileOutputVectorManager` class is used to record output vectors, there are two more options to fine-tune its resource usage. `output-vectors-memory-limit` specifies the total memory that can be used for buffering output vectors. Larger values produce less fragmented vector files (i.e. cause vector data to be grouped into larger chunks), and therefore allow more efficient processing later. `vector-max-buffered-values` specifies the maximum number of values to buffer per vector, before writing out a block into the output vector file. The default is no per-vector limit (i.e. only the total memory limit is in effect.)

12.2.7 Saving Parameters as Scalars

When you are running several simulations with different parameter settings, you will usually want to refer to selected input parameters in the result analysis as well – for example when drawing a throughput (or response time) versus load (or network background traffic) plot. Average throughput or response time numbers are saved into the output scalar files, and it is useful for the input parameters to get saved into the same file as well.

For convenience, OMNEST automatically saves the iteration variables into the output scalar file if they have numeric value, so they can be referred to during result analysis.

WARNING: If an iteration variable has non-numeric value, it will not be recorded automatically and cannot be used during analysis. This can happen unintentionally if you specify units inside an iteration variable list:

```
**.param = exponential( ${mean=0.2s, 0.4s, 0.6s} )   #WRONG!
**.param = exponential( ${mean=0.2, 0.4, 0.6}s )     #OK
```

Module parameters can also be saved, but this has to be requested by the user, by configuring `param-record-as-scalar=true` for the parameters in question. The configuration key is a pattern that identifies the parameter, plus `.param-record-as-scalar`. An example:

```
**.host[*].networkLoad.param-record-as-scalar = true
```

This looks simple enough, however there are three pitfalls: non-numeric parameters, too many matching parameters, and random-valued volatile parameters.

First, the scalar file only holds numeric results, so non-numeric parameters cannot be recorded – that will result in a runtime error.

Second, if wildcards in the pattern match too many parameters, that might unnecessarily increase the size of the scalar file. For example, if the `host[]` module vector size is 1000 in the example below, then the same value (3) will be saved 1000 times into the scalar file, once for each host.

```
**.host[*].startTime = 3
**.host[*].startTime.param-record-as-scalar = true  # saves "3" once for each host
```

Third, recording a random-valued volatile parameter will just save a random number from that distribution. This is rarely what you need, and the simulation kernel will also issue a warning if this happens.

```
**.interarrivalTime = exponential(1s)
**.interarrivalTime.param-record-as-scalar = true  # wrong: saves random values!
```

These pitfalls are quite common in practice, so it is usually better to rely on the iteration variables in the result analysis. That is, one can rewrite the above example as

```
**.interarrivalTime = exponential( ${mean=1}s )
```

and refer to the `$mean` iteration variable instead of the interarrivalTime module parameter(s) during result analysis. `param-record-as-scalar=true` is not needed, because iteration variables are automatically saved into the result files.

12.2.8 Recording Precision

Output scalar and output vector files are text files, and floating point values (`double`s) are recorded into it using `fprintf()`'s `"%g"` format. The number of significant digits can be configured using the **`output-scalar-precision`** and **`output-vector-precision`** configuration options.

The default precision is 12 digits. The following has to be considered when setting a different value:

IEEE-754 doubles are 64-bit numbers. The mantissa is 52 bits, which is roughly equivalent to 16 decimal places (52*log(2)/log(10)). However, due to rounding errors, usually only 12..14 digits are correct, and the rest is pretty much random garbage which should be ignored. However, when you convert the decimal representation back into a `double` for result processing, an additional small error will occur, because 0.1, 0.01, etc. cannot be accurately represented in binary. This conversion error is usually smaller than what that the `double` variable already had before recording into the file. However, if it is important, you can eliminate this error by setting the recording precision to 16 digits or more (but again, be aware that the last digits are garbage). The practical upper limit is 17 digits, setting it higher doesn't make any difference in `fprintf()`'s output.

Errors resulting from converting to/from decimal representation can be eliminated by choosing an output vector/output scalar manager class which stores `double`s in their native binary form. The appropriate configuration options are **`outputvectormanager-class`** and **`outputvectormanager-class`**. For example, `cMySQLOutputScalarManager` and `cMySQLOutputScalarManager` provided in `samples/database` fulfill this requirement.

However, before worrying too much about rounding and conversion errors, consider the *real* accuracy of your results:

- In real life, it is very difficult to measure quantities (weight, distance, even time) with more than a few digits of precision. What precision are your input data? For example, if you approximate inter-arrival time as *exponential(0.153)* when the mean is really *0.152601...* and the distribution is not even exactly exponential, you are already starting out with a bigger error than rounding can cause.

- The simulation model is itself an approximation of real life. How much error do the (known and unknown) simplifications cause in the results?

12.3 Result Files

12.3.1 The OMNEST Result File Format

By default, OMNEST saves simulation results into textual, line-oriented files. The advantage of a text-based, line-oriented format is that it is very accessible and easy to parse with a wide range of tools and languages, and still provides enough flexibility to be able to represent the data it needs to (in contrast to e.g. CSV). This section provides an overview of these file formats (output vector and output scalar files); the precise specification is available in the Appendix (J).

By default, each file contains data from one run only.

Result files start with a header that contains several attributes of the simulation run: a reasonably globally unique run ID, the network NED type name, the experiment-measurement-replication labels, the values of iteration variables and the repetition counter, the date and time, the host name, the process id of the simulation, random number seeds, configuration options, and so on. These data can be useful during result processing, and increase the reproducibility of the results.

Vectors are recorded into a separate file for practical reasons: vector data usually consume several magnitudes more disk space than scalars.

Output Vector Files

All output vectors from a simulation run are recorded into the same file. The following sections describe the format of the file, and how to process it.

An example file fragment (without header):

```
...
vector 1   net.host[12]  responseTime  TV
1  12.895  2355.66
1  14.126  4577.66664666
vector 2   net.router[9].ppp[0] queueLength  TV
2  16.960  2
1  23.086  2355.66666666
2  24.026  8
...
```

There two types of lines: vector declaration lines (beginning with the word `vector`), and data lines. A *vector declaration line* introduces a new output vector, and its columns are: vector Id, module of creation, name of `cOutVector` object, and multiplicity (usually 1). Actual data recorded in this vector are on *data lines* which begin with the vector Id. Further columns on data lines are the simulation time and the recorded value.

Since OMNEST 4.0, vector data are recorded into the file clustered by output vectors, which, combined with index files, allows much more efficient processing. Using the index file, tools can extract particular vectors by reading only those parts of the file where the desired data are located, and do not need to scan through the whole file linearly.

Scalar Result Files

Fragment of an output scalar file (without header):

```
...
scalar "lan.hostA.mac" "frames sent"  99
scalar "lan.hostA.mac" "frames rcvd"  3088
scalar "lan.hostA.mac" "bytes sent"   64869
scalar "lan.hostA.mac" "bytes rcvd"   3529448
...
```

Every scalar generates one `scalar` line in the file.

Statistics objects (`cStatistic` subclasses such as `cStdDev`) generate several lines: mean, standard deviation, etc.

12.3.2 SQLite Result Files

Starting from version 5.1, OMNEST contains experimental support for saving simulation results into SQLite database files. The perceived advantage of SQLite is existing support in many existing tools and languages (no need to write custom parsers), and being able to use the power of the SQL language for queries. The latter is very useful for processing scalar results, and less so for vectors and histograms.

To let a simulation record its results in SQLite format, add the following configuration options to its `omnetpp.ini`:

```
outputvectormanager-class="omnetpp::envir::SqliteOutputVectorManager"
outputscalarmanager-class="omnetpp::envir::SqliteOutputScalarManager"
```

NOTE: Alternatively, to make SQLite the default format, recompile OMNEST with `PREFER_SQLITE_RESULT_FILES=yes` set in `configure.user`. (Don't forget to also run `./configure` before `make`.)

The SQLite result files will be created with the same names as textual result files. The two formats also store exactly the same data, only in a different way (there is one-to-one correspondence between them.) The Simulation IDE and `scavetool` also understand both formats.

HINT: If you want to get acquainted with the organization of SQLite result files, exploring one in a graphical tool such as SQLiteBrowser or SQLite Studio should be a good start.

The database schema can be found in Appendix J.

12.3.3 Scavetool

OMNEST's `opp_scavetool` program is a command-line tool for exploring, filtering and processing of result files, and exporting the result in formats digestible by other tools.

Commands

`opp_scavetool`'s functionality is grouped under four commands: `query`, `export`, `index`, and `help`.

- **query**: Query the contents of result files. One can list runs, run attributes, result items, unique result names, unique module names, unique configuration names, etc. One can filter for result types (scalar/vector/histogram), and by run, module name, result name and value, using match expressions. There are various options controlling the format of the output (group-by-runs; grep-friendly; suppress labels; several modes for identifying the run in the output, etc.)
- **export**: Export results in various formats. Results can be filtered by run, module name, result name and more, using match expressions. Output vectors can be cropped to a time interval. Several output formats are available: CSV in two flavours (one for machine consumption, and a more informal one for human consumption via loading into spreadsheet programs), OMNEST output scalar/vector file (default), OMNEST SQLite result file, and JSON (again two flavours: one strictly adhering to the JSON rules, and another one with slightly more relaxed rules but being also more expressive). All exporters have multiple options for fine-tuning the output.
- **index**: Generate index files (.vci) for vector files. Note that this command is usually not needed, as other scavetool commands automatically create vector file indices if they are missing or out of date (unless indexing is explicitly disabled.) This command can also be used to rebuild a vector file so that data are clustered by vectors for more efficient access.
- **help**: Prints help. The synopsys is `opp_scavetool help <topic>`, where any command name can be used as topic, plus there are additional ones like `patterns` or `filters`. `scavetool <command> -h` also works.

The default command is `query`, so its name may be omitted on the command line.

Examples

The following example prints a one-line summary about the contents of result files in the current directory:

```
$ opp_scavetool *.sca *.vec
runs: 42   scalars: 294  parameters: 7266  vectors: 22  statistics: 0  ...
```

Listing all results is possible with `-l`:

```
$ opp_scavetool -l *.sca *.vec
PureAlohaExperiment-439-20161216-18:56:20-27607:

scalar Aloha.server  duration              26.3156
scalar Aloha.server  collisionLength:mean  0.139814
vector Aloha.host[0] radioState:vector vectorId=2 count=3 mean=0.33 ..
vector Aloha.host[1] radioState:vector vectorId=3 count=9 mean=0.44 ..
vector Aloha.host[2] radioState:vector vectorId=4 count=5 mean=0.44 ..
...
```

To export all scalars in CSV, use the following command:

```
$  opp_scavetool export -F CSV-R -o x.csv *.sca
Exported 294 scalars, 7266 parameters, 84 histograms
```

The next example writes the queueing and transmission time vectors of `sink` modules into a CSV file.

```
$ opp_scavetool export -f 'module=~**.sink AND ("queueing time" OR "tx time")'
  -o out.csv -F CSV-R *.vec
Exported 15 vectors
```

12.4 Result Analysis

The recommended way of analyzing simulation results is using the *Analysis Tool* in the *Simulation IDE*. The Analysis Tool provides a comfortable user interface for selecting result files to work with, browsing their contents, selecting results of interest from them, and creating plots. The resulting plots and their underlying data can be exported in several formats, both for individual charts and in batches. You can choose from several chart types, and new ones can also be created.

Charts in the Analysis Tool are powered by Python. The Python scripts underlying the various charts are open for the user to view and edit, so arbitrary logic and computations can be implemented. Visualization may use the IDE's native plotting widgets, but it can also be done with Matplotlib. The use of Matplotlib allows virtually limitless possibilities for visualization.[1] The IDE's own plotting widgets are more limited in functionality, but they are much more scalable than Matplotlib.

> **NOTE:** Note the terminology. Although the nouns *chart* and *plot* are almost interchangeable in everyday speech, we assign related but clearly distinct meanings to them when discussing OMNEST result analysis. By *chart* we essentially mean a Python script with its associated metadata and parameterization that serves as a "recipe" for producing a plot; and the word *plot* is used to refer to the graphics which appears as the result of running said script.

Chart scripts can also be used outside the IDE. Those saved as part of the IDE's analysis files (`.anf`) can be viewed or run using the `opp_chartool` command-line program. You can also take advantage of result processing capabilities in standalone Python scripts. When chart scripts run outside the IDE, native plotting widgets are "emulated" using Matplotlib.

The Analysis Tool is covered in detail in the User Guide. The following sections deal with the programming API.

12.4.1 Python Packages

Chart scripts heavily build on the following, fairly standard Python packages:

- *NumPy*: We use NumPy for its efficient representation of numeric arrays and the operations on them.
- *Pandas*: Pandas `DataFrame`s are used for representing and manipulating simulation results.
- *Matplotlib*: For creating the actual plots.

OMNEST adds the following packages:

[1] Note that Matplotlib also has extensions like Seaborn, Canopy, HoloViews, etc., which can also be used in chart scripts, further expanding the set of possibilities.

- *omnetpp.scave.results*: Provides access to the simulation results for the chart script. The results are returned as Pandas `DataFrame`s of various formats.
- *omnetpp.scave.chart*: Provides access to the properties of the current chart for the chart script.
- *omnetpp.scave.ideplot*: This module is the interface for displaying plots with the IDE's native plotting widgets. The API is intentionally very close to `matplotlib.pyplot`, which facilitates porting scripts across the two APIs. When a chart script runs outside the context of a native plotting widget, such as when run from `opp_charttool`, the functions are emulated with Matplotlib.
- *omnetpp.scave.utils*: A collection of utility function for data manipulation and plotting, built on top of `DataFrame`s and the `chart` and `plot` packages from `omnetpp.scave`.
- *omnetpp.scave.vectorops*: Contains operations that can be applied to output vectors.

An additional module:

- *omnetpp.scave.analysis*: Provides support for reading and writing analysis (anf) files from Python, and running chart scripts in them for display, image export or data export.

These packages are documented in detail in Appendix L.

12.4.2 An Example Chart Script

Since information on NumPy, Pandas and Matplotlib can be found in abundance online, and a reference of the `omnetpp.scave.*` Python packages is in the Appendix, it makes little sense here to go through the functionality they provide. Instead, in this section we walk through an actual chart script in order to see how it looks in practice.

The chosen chart script is that of the bar chart. It is quite representative, so it will prepare you to understand other chart scripts, modify them for custom needs, or write your own; yet it is short and simple, so it is easy to follow. We are going to list the source, and pause for explanations after every few lines.

```
from omnetpp.scave import results, chart, utils
```

The first lines is for importing the packages we are going to use. This is necessary because nothing is imported by default when the chart script starts.

Notice how all imported modules are under the `omnetpp.scave` module, and not directly from `numpy` or `pandas` or `matplotlib` packages. This is because almost all necessary functionality is already encompassed in convenience methods in (mostly) the `utils` and `plot` modules.

```
# get chart properties
props = chart.get_properties()
```

Then, we obtain the properties of the bar chart from the `chart` module. The `props` object we get is a Python `dict` whose entries can be influnced by the chart properties dialog, and they serve as parameters for the chart script and the resulting plot.

Try adding `print(props)` to the code, or `for k,v in props.items(): print(repr(k), "=", repr(v))` for fancier output. After the chart script ran, you should see an output similar to the following:

```
'confidence_level' = '95\%'
'filter' = 'type =~ scalar AND name =~ channelUtilization:last'
'grid_show' = 'true'
'legend_prefer_result_titles' = 'true'
'title' = ''
'legend_show' = 'true'
'matplotlibrc' = ''
...
```

Many entries should look familiar. Indeed, most of the entries have direct correspondence to widgets in the *Chart Properties* dialog in the IDE. Note that all values are strings.

```
utils.preconfigure_plot(props)
```

The `preconfigure_plot()` call is a mandatory part of a chart script, and its job is to ensure that visual properties take effect in the plot. Note that we will also have a `postconfigure_plot()` call, because some properties must be set before, and others after the plotting.

```
# collect parameters for query
filter_expression = props["filter"]
include_fields = props["include_fields"] == "true"
```

Here we obtain the result query string from the properties. The query string selects the subset of the results that serve as input to the chart from the set of all results loaded from the result files. The `"filter"` property is common to almost all chart types.

Since bar charts work with scalars, we provide the user an opportunity to select whether the fields (such as `:mean`, `:count`, `:sum`, etc.) of vector, statistics, and histogram results should also be included in the source dataset, as scalars.

```
# query scalar data into dataframe
try:
    df = results.get_scalars(filter_expression, include_fields=include_fields,
             include_attrs=True, include_runattrs=True, include_itervars=True)
except ValueError as e:
    raise chart.ChartScriptError("Error while querying results: " + str(e))
```

Here, `results.get_scalars()` is the most important part. It uses the `results` module to get the data for the plot. The resulting Pandas `DataFrame` will have one row for each scalar result. Columns include `runID` which uniquely identifies the simulation run, `module`, `name` and `value` which refer to the scalar, and many other columns that represent metadata such as result attributes, iteration variables and run attributes: `iaMean`, `numHosts`, `configname`, `datetime`, etc.

Try `print(df)` to print the dataframe contents. You will get something like this (for brevity, we dropped the less important columns from the output, and abbreviated the name of the last colum from `repetition`):

```
          module                      name     value iaMean numHosts rep.
0   Aloha.server  channelUtilization:last  0.156057      1       10    0
1   Aloha.server  channelUtilization:last  0.156176      1       10    1
2   Aloha.server  channelUtilization:last  0.196381      2       10    0
3   Aloha.server  channelUtilization:last  0.193253      2       10    1
4   Aloha.server  channelUtilization:last  0.176507      3       10    0
5   Aloha.server  channelUtilization:last  0.176136      3       10    1
6   Aloha.server  channelUtilization:last  0.152471      4       10    0
```

```
7   Aloha.server  channelUtilization:last  0.154667      4        10    1
11  Aloha.server  channelUtilization:last  0.108992      7        10    0
...
```

Also note in the snippet above that `try...except` was used to catch exceptions (usually syntax errors in the query), and gracefully report them back to the user instead of letting a stack trace appear in the console. Raising a `chart.ChartScriptError` displays the passed message in the plot area.

```
if df.empty:
    raise chart.ChartScriptError("The result filter returned no data.")
```

If the query matched nothing, let the user know instead of letting them figure out from the empty plot they get, again, by raising an exception of the appropriate type.

```
groups, series = utils.select_groups_series(df, props)
```

The user may fill in the *Groups* and *Series* fields of the *Chart Properties* dialog to control how to organize the bar chart. As each field may contain multiple variables (separated by comma), we split the values to convert them to lists.

If these fields (properties) are left empty, the script tries to find reasonable values for them. It also detects various misconfiguration cases (such as nonexisting column names given in, or overlap between, the "groups" and "series" columns), and report them back to the user. Leaving out these checks would, in most cases, cause those errors to manifest themselves in later steps as spurious Pandas exceptions, whose wording often provides little guidance to the user about what actually went wrong.

```
confidence_level = utils.get_confidence_level(props)
```

Extract the confidence level requested by the user from the properties. (`"none"` is the string that the user can select from the combo box in the dialog to turn off confidence interval computation.)

```
valuedf, errorsdf, metadf =
    utils.pivot_for_barchart(df, groups, series, confidence_level)
utils.plot_bars(valuedf, errorsdf, metadf, props)
```

At last, we arrive at the important part of the script, which is pivoting and plotting the data. The `utils.pivot_for_barchart()` function is used for pivoting, and `utils.plot_bars()` for plotting.

If you add a `print(valuedf)` statement, you can see the result of pivoting:

```
numHosts        10        15        20
iaMean
1         0.156116  0.089539  0.046586
2         0.194817  0.178159  0.147564
3         0.176321  0.191571  0.183976
4         0.153569  0.182324  0.190452
5         0.136997  0.168780  0.183742
7         0.109281  0.141556  0.164038
9         0.089658  0.120800  0.142568
```

If the user requested no confidence interval (error bars), `errorsdf` will be `None`.

In this case, the default 95% is used, so `print(errorsdf)` shows this:

```
numHosts        10        15        20
iaMean
1          0.000117  0.001616  0.001968
2          0.003065  0.000619  0.002162
3          0.000364  0.001426  0.001704
4          0.002152  0.000918  0.002120
5          0.002391  0.000411  0.000625
7          0.000568  0.001729  0.002221
9          0.001621  0.002385  0.000259
```

This dataframe has the exact same structure (column and row headers) as `valuedf`, only the values are different - they are the half-length of the confidence interval corresponding to the selected confidence level, so it can be interpreted as a "+/-" range.

The third dataframe, `metadf` contains various pieces of metadata about the results,

Here are just a few columns from it:

```
                        measurement        module                       title
iaMean
1      $numHosts=10, $iaMean=1, etc.  Aloha.server  channel utilization, last
2      $numHosts=10, $iaMean=2, etc.  Aloha.server  channel utilization, last
3      $numHosts=10, $iaMean=3, etc.  Aloha.server  channel utilization, last
4      $numHosts=10, $iaMean=4, etc.  Aloha.server  channel utilization, last
5      $numHosts=10, $iaMean=5, etc.  Aloha.server  channel utilization, last
7      $numHosts=10, $iaMean=7, etc.  Aloha.server  channel utilization, last
9      $numHosts=10, $iaMean=9, etc.  Aloha.server  channel utilization, last
```

This dataframe is used to assemble the legend labels for the series of bars on the plot. It has the same row headers as `valuedf`, but the column headers are the names of run and result attributes, and iteration variables. Where multiple different values would have had to be put into the same cell, only the first one is present, and an "etc." is appended.

Note that in some other types of charts (line charts, histogram charts, etc.), separating the results into separate dataframes like this is not necessary, as those charts do not do pivot operations on their results, and the corresponding plots accept data in formats that can hold the metadata in the same dataframe as the main values to be plotted.

The drawn plot is shown in Figure 12.1:

```
utils.postconfigure_plot(props)
```

This line applies the rest of the visual properties to the plot.

```
utils.export_image_if_needed(props)
utils.export_data_if_needed(df, props)
```

These lines perform image and data export. Exporting is done by running chart scripts with certain properties set in order to indicate to the chart script that exporting is requested.

The `utils.export_image_if_needed()` and `utils.export_data_if_needed()` functions take those flag properties and a lot of other properties related to exporting. The latter saves the dataframe given to it as argument.

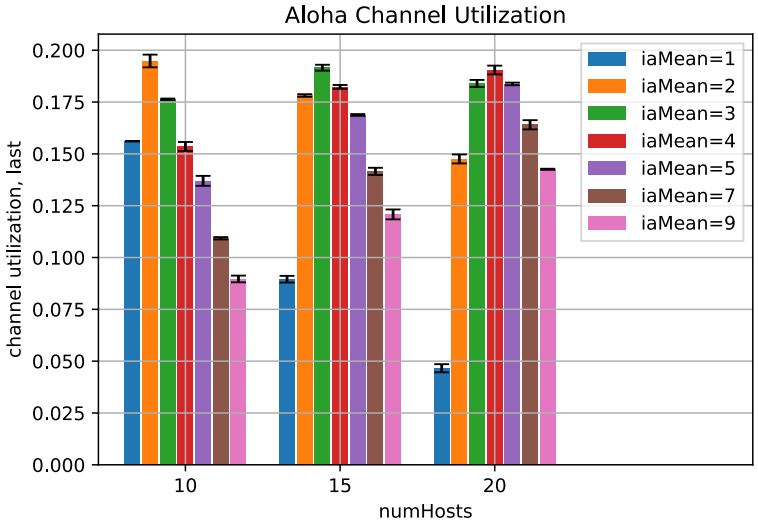


Figure 12.1: The resulting bar plot, showing confidence interval as error bars

12.5 Alternatives

Based on your personal preferences, you may opt to use to a different environment, language or tool than the IDE's Analysis tool for analysing simulation results. Here are some of the possibilities:

- Use your favourite Python editor to write the analysis scripts, using the packages mentioned in the previous section.
- A *Jupyter Notebook* can also be used to write up the analysis steps, still using Python and the above packages.
- If your simulations produce a huge amount of data, you might prefer using the SQLite result file format which allows you to run queries without loading all data into memory. Python also has packages to access SQLite files, e.g. `sqlite3`.
- If you prefer *GNU R* to Python/Pandas, you might use that.
- Or you may go for *MATLAB* or *GNU Octave* if you feel at home with them.
- *Spreadsheet* programs such as Microsoft Excel might be suitable if the amount of data allows it. One drawback of using spreadsheets is the manual work associated with preparing and reloading data every time simulations are re-run.
- A dedicated visual analytics environment such as *Tableau* might be a better choice than spreadsheets.

For environments where reading OMNeT++ result files or SQLite result files is not a real possibility, probably the easiest way to go is to export simulation results into CSV with `opp_scavetool`. CSV is a universal format that nearly all tools understand.

Chapter 13

Eventlog

13.1 Introduction

The eventlog feature and related tools have been added to OMNEST with the aim of helping the user understand complex simulation models and correctly implement the desired component behaviors. Using these tools, one can examine details of recorded history of a simulation, focusing on the behavior instead of the statistical results.

The eventlog file is created automatically during a simulation run upon explicit request configurable in the ini file. The resulting file can be viewed in the OMNEST IDE using the Sequence Chart and the Eventlog Table or can be processed by the command line Eventlog Tool. These tools support filtering the collected data to allow you to focus on events relevant to what you are looking for. They allow examining causality relationships and provide filtering based on simulation times, event numbers, modules and messages.

The simulation kernel records into the eventlog among others: user level messages, creation and deletion of modules, gates and connections, scheduling of self messages, sending of messages to other modules either through gates or directly, and processing of messages (that is events). Optionally, detailed message data can also be automatically recorded based on a message filter. The result is an eventlog file which contains detailed information of the simulation run and later can be used for various purposes.

NOTE: The eventlog file may become quite large for long-running simulations (often hundreds of megabytes, but occasionally several gigabytes), especially when message detail recording is turned on.

13.2 Configuration

To record an eventlog file during the simulation, insert the following line into the ini file:

```
record-eventlog = true
```

NOTE: Eventlog recording is turned off by default, because creating the eventlog file might significantly decrease the overall simulation performance.

13.2.1 File Name

The simulation kernel will write the eventlog file during the simulation into the file specified by the following ini file configuration entry (showing the default file name pattern here):

```
eventlog-file = ${resultdir}/${configname}-${runnumber}.elog
```

13.2.2 Recording Intervals

The size of an eventlog file is approximately proportional to the number of events it contains. To reduce the file size and speed up the simulation, it might be useful to record only certain events. The `eventlog-recording-intervals` configuration option instructs the kernel to record events only in the specified intervals. The syntax is similar to that of `vector-recording-intervals`.

An example:

```
eventlog-recording-intervals = ..10.2, 22.2..100, 233.3..
```

13.2.3 Recording Modules

Another factor that affects the size of an eventlog file is the number of modules for which the simulation kernel records events during the simulation. The `module-eventlog-recording` per-module configuration option instructs the kernel to record only the events that occurred in the matching modules. The default is to record events from all modules. This configuration option only applies to simple modules.

The following example records events from any of the routers whose index is between 10 and 20, and turns off recording for all other modules.

```
**.router[10..20].**.module-eventlog-recording = true
**.module-eventlog-recording = false
```

13.2.4 Recording Message Data

Since recording message data dramatically increases the size of the eventlog file and also slows down the simulation, it is turned off by default, even if writing the eventlog is enabled. To turn on message data recording, supply a value for the `eventlog-message-detail-pattern` option in the ini file.

An example configuration for an IEEE 80211 model that records the `encapsulationMsg` field and all other fields whose name ends in `Address`, from messages whose class name ends in `Frame` looks like this:

```
eventlog-message-detail-pattern = *Frame:encapsulatedMsg,*Address
```

An example configuration for a TCP/IP model that records the port and address fields in all network packets looks like the following:

```
eventlog-message-detail-pattern =
 PPPFrame:encapsulatedPacket|IPDatagram:encapsulatedPacket,*Address|TCPSegment:*Po
```

13.3 Eventlog Tool

The Eventlog Tool is a command line tool to process eventlog files. Invoking it without parameters will display usage information. The following are the most useful commands for users.

13.3.1 Filter

The eventlog tool provides off line filtering that is usually applied to the eventlog file after the simulation has been finished and before actually opening it in the OMNEST IDE or processing it by any other means. Use the filter command and its various options to specify what should be present in the result file.

13.3.2 Echo

Since the eventlog file format is text based and users are encouraged to implement their own filters, a way is needed to check whether an eventlog file is correct. The echo command provides a way to check this and help users creating custom filters. Anything not echoed back by the eventlog tool will not be taken into consideration by the other tools found in the OMNEST IDE.

NOTE: Custom filter tools should filter out whole events only, otherwise the consequences are undefined.

Chapter 14

Documenting NED and Messages

14.1 Overview

OMNEST provides a tool which can generate HTML documentation from NED files and message definitions. Like Javadoc and Doxygen, the NED documentation tool makes use of source code comments. The generated HTML documentation lists all modules, channels, messages, etc., and presents their details including description, gates, parameters, assignable submodule parameters, and syntax-highlighted source code. The documentation also includes clickable network diagrams (exported from the graphical editor) and usage diagrams as well as inheritance diagrams.

The documentation tool integrates with Doxygen, meaning that it can hyperlink simple modules and message classes to their C++ implementation classes in the Doxygen documentation. If the C++ documentation is generated with some Doxygen features turned on (such as *inline-sources* and *referenced-by-relation*, combined with *extract-all*, *extract-private* and *extract-static*), the result is an easily browsable and very informative presentation of the source code.

NED documentation generation is available as part of the OMNEST IDE, and also as a command-line tool (`opp_neddoc`).

14.2 Documentation Comments

Documentation is embedded in normal comments. All `//` comments that are in the "right place" (from the documentation tool's point of view) will be included in the generated documentation. [1]

Example:

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//
simple Gen
```

[1] In contrast, Javadoc and Doxygen use special comments (those beginning with `/**`, `///`, `//<` or a similar marker) to distinguish documentation from "normal" comments in the source code. In OMNEST there is no need for that: NED and the message syntax is so compact that practically all comments one would want to write in them can serve documentation purposes.

```
{
    parameters:
        string destAddress;  // destination MAC address
        int protocolId;      // value for SSAP/DSAP in Ethernet frame
        double waitMean @unit(s); // mean for exponential interarrival times
    gates:
        output out;  // to Ethernet LLC
}
```

One can also place comments above parameters and gates, which is better suited for long explanations. Example:

```
//
// Deletes packets and optionally keeps statistics.
//
simple Sink
{
    parameters:
        // Turns statistics generation on/off. This is a very long
        // comment because it has to be described what statistics
        // are collected.
        bool collectStatistics = default(true);
    gates:
        input in;
}
```

14.2.1 Private Comments

Lines that start with `//#` will not appear in the generated documentation. Such lines can be used to make "private" comments like `FIXME` or `TODO`, or to comment out unused code.

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//# TODO above description needs to be refined
//
simple Gen
{
    parameters:
        string destAddress;  // destination MAC address
        int protocolId;      // value for SSAP/DSAP in Ethernet frame
        //# double burstiness;  -- not yet supported
        double waitMean @unit(s); // mean for exponential interarrival times
    gates:
        output out;  // to Ethernet LLC
}
```

14.2.2 More on Comment Placement

Comments should be written where the tool will find them. This is a) immediately above the documented item, or b) after the documented item, on the same line.

In the former case, make sure there is no blank line left between the comment and the documented item. Blank lines detach the comment from the documented item.

Example:

```
// This is wrong! Because of the blank line, this comment is not
// associated with the following simple module!

simple Gen
{
    ...
}
```

Do not try to comment groups of parameters together. The result will be awkward.

14.3 Referring to Other NED and Message Types

One can reference other NED and message types by name in comments. There are two styles in which references can be written: automatic linking and tilde linking. The same style must be following throughout the whole project, and the correct one must be selected in the documentation generator tool when it is run.

14.3.1 Automatic Linking

In the automatic linking style, words that match existing NED of message types are hyperlinked automatically. It is usually enough to write the simple name of the type (e.g. `TCP`), one does not need to spell out the fully qualified type (`inet.transport.tcp.TCP`), although that is also allowed.

Automatic hyperlinking is sometimes overly agressive. For example, when the words *IP address* appear in a comment and the project contains an `IP` module, it will create a hyperlink to the module, which is not desirable. One can prevent hyperlinking of a word by inserting a backslash in front it: `\IP address`. The backslash will not appear in the HTML output. The `<nohtml>` tag will also prevent hyperlinking words in the enclosed text: `<nohtml>IP address</nohtml>`. On the other hand, if a backslash needs to be printed immediately in front of a word (e.g. output *"use \t to print a Tab"*), use either two backslashes (`use \\t...`) or the `<nohtml>` tag (`<nohtml>use \t...</nohtml>`). Backslashes in other contexts (i.e. when not in front of a word) do not have a special meaning, and are preserved in the output.

The detailed rules:

1. Words matching a type name are automatically hyperlinked
2. A backslash immediately followed by an identifier (i.e. letter or underscore) prevents hyperlinking, and the backslash is removed from the output
3. A double backslash followed by an identifier produces a single backslash, plus the potentially hyperlinked identifier
4. Backslashes in any other contexts are not interpreted, and preserved in the output
5. Tildes are not interpreted, and are preserved in the output
6. Inside `<nohtml>`, no backslash processing or hyperlinking takes place

14.3.2 Tilde Linking

In the tilde style, only words that are explicitly marked with a tilde are subject to hyperlinking: `~TCP`, `~inet.transport.tcp.TCP`.

To produce a literal tilde followed by an identifier in the output (for example, to output *"the ~TCP() destructor"*), the tilde character needs to be doubled: `the ~~TCP() destructor`.

The detailed rules:

1. Words matching a type name are *not* hyperlinked automatically
2. A tilde immediately followed by an identifier (i.e. letter or underscore) will be hyperlinked, and the tilde is removed from the output. It is considered an error if there is no type with that name.
3. A double tilde followed by an identifier produces a single tilde plus the identifier
4. Tildes in any other contexts are not interpreted, and preserved in the output
5. Backslashes are not interpreted, and are preserved in the output
6. Inside `<nohtml>`, no tilde processing or hyperlinking takes place

14.4 Text Layout and Formatting

14.4.1 Paragraphs and Lists

When writing documentation comments longer than a few sentences, one often needs structuring and formatting facilities. NED provides paragraphs, bulleted and numbered lists, and basic formatting support. More sophisticated formatting can be achieved using HTML.

Paragraphs can be created by separating text by blanks lines. Lines beginning with "`-`" will be turned into bulleted lists, and lines beginning with "`-#`" into numbered lists. An example:

```
//
// Ethernet MAC layer. MAC performs transmission and reception of frames.
//
// Processing of frames received from higher layers:
// - sends out frame to the network
// - no encapsulation of frames -- this is done by higher layers.
// - can send PAUSE message if requested by higher layers (PAUSE protocol,
//   used in switches). PAUSE is not implemented yet.
//
// Supported frame types:
// -# IEEE 802.3
// -# Ethernet-II
//
```

14.4.2 Special Tags

The documentation tool understands the following tags and will render them accordingly: `@author`, `@date`, `@todo`, `@bug`, `@see`, `@since`, `@warning`, `@version`. Example usage:

```
//
// @author Jack Foo
// @date 2005-02-11
//
```

14.4.3 Text Formatting Using HTML

Common HTML tags are understood as formatting commands. The most useful tags are: `<i>..</i>` (italic), `<b>..</b>` (bold), `<tt>..</tt>` (typewriter font), `<sub>..</sub>` (subscript), `<sup>..</sup>` (superscript), `<br>` (line break), `<h3>` (heading), `<pre>..</pre>` (preformatted text) and `<a href=..>..</a>` (link), as well as a few other tags used for table creation (see below). For example, `<i>Hello</i>` will be rendered as "*Hello*" (using an italic font).

The complete list of HTML tags interpreted by the documentation tool are: `<a>`, `<b>`, `<body>`, `<br>`, `<center>`, `<caption>`, `<code>`, `<dd>`, `<dfn>`, `<dl>`, `<dt>`, `<em>`, `<form>`, `<font>`, `<hr>`, `<h1>`, `<h2>`, `<h3>`, `<i>`, `<input>`, `<img>`, `<li>`, `<meta>`, `<multicol>`, `<ol>`, `<p>`, `<small>`, `<span>`, `<strong>`, `<sub>`, `<sup>`, `<table>`, `<td>`, `<th>`, `<tr>`, `<tt>`, `<kbd>`, `<ul>`, `<var>`.

Any tags not in the above list will not be interpreted as formatting commands but will be printed verbatim – for example, `<what>bar</what>` will be rendered literally as "<what>bar</what>" (unlike HTML where unknown tags are simply ignored, i.e. HTML would display "bar").

With links to external pages or web sites, its useful to add the `target="_blank"` attribute to ensure pages come up in a new browser tab, and not in the current frame. Alternatively, one can use the `target="_top"` attribute which replaces all frames in the current browser.

Examples:

```
//
// For more info on Ethernet and other LAN standards, see the
// <a href="http://www.ieee802.org/" target="_blank">IEEE 802
// Committee's site</a>.
//
```

One can also use the `<a href=..>` tag to create links within the page:

```
//
// See the <a href="#resources">resources</a> in this page.
// ...
// <a name="resources"><b>Resources</b></a>
// ...
//
```

One can use the `<pre>..</pre>` HTML tag to insert source code examples into the documentation. Line breaks and indentation will be preserved, but HTML tags continue to be interpreted (they can be turned off with `<nohtml>`, see later).

Example:

```
// <pre>
// // my preferred way of indentation in C/C++ is this:
// <b>for</b> (<b>int</b> i = 0; i < 10; i++) {
//     printf(<i>"%d\n"</i>, i);
// }
// </pre>
```

will be rendered as

```
// my preferred way of indentation in C/C++ is this:
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

HTML is also the way to create tables. The example below

```
//
// <table border="1">
//   <tr>  <th>#</th> <th>number</th> </tr>
//   <tr>  <td>1</td> <td>one</td>    </tr>
//   <tr>  <td>2</td> <td>two</td>    </tr>
//   <tr>  <td>3</td> <td>three</td>  </tr>
// </table>
//
```

will be rendered approximately as:

#	number
1	one
2	two
3	three

14.4.4 Escaping HTML Tags

In some cases, one needs to turn off interpreting HTML tags (`<i>`, `<b>`, etc.) as formatting, and rather include them as literal text in the generated documentation. This can be achieved by surrounding the text with the `<nohtml>...</nohtml>` tag. For example,

```
// Use the <nohtml><i></nohtml> tag (like <tt><nohtml><i>this</i></nohtml><tt>)
// to write in <i>italic</i>.
```

will be rendered as "Use the **<i>** tag (like `<i>this</i>`) to write in *italic*."

`<nohtml>...</nohtml>` will also prevent `opp_neddoc` from hyperlinking words that are accidentally the same as an existing module or message name. Prefixing the word with a backslash will achieve the same. That is, either of the following will do:

```
// In <nohtml>IP</nohtml> networks, routing is...
```

```
// In \IP networks, routing is...
```

Both will prevent hyperlinking the word *IP* in case there is an `IP` module in the project.

14.5 Incorporating Extra Content

14.5.1 Adding a Custom Title Page

The title page is the one that appears in the main frame after opening the documentation in the browser. By default, it contains a boilerplate text with the title *"OMNEST Model Documen-*

tation". Model authors will probably want to customize that, and at least change the title be more specific.

A title page is defined with a `@titlepage` directive. It needs to appear in a file-level comment.

> **NOTE:** A file-level comment is one that appears at the top of a NED file, and is separated from any other NED content by at least one *blank line*.

While one can place the title page definition into any NED or MSG file, it is probably a good idea to create a dedicated NED file for it. Lines up to the next `@page` line or the end of the comment (whichever comes first) are interpreted as part of the page.

The page should start with a title, as the documentation tool doesn't add one. Use the `<h1>..</h1>` HTML tag for that.

Example:

```
//
// @titlepage
// <h1>Ethernet Model Documentation</h1>
//
// This documents the Ethernet model created by David Wu and refined by Andras
// Varga at CTIE, Monash University, Melbourne, Australia.
//
```

14.5.2 Adding Extra Pages

One can add new pages to the documentation using the `@page` directive. `@page` may appear in any file-level comment, and has the following syntax:

```
// @page filename.html, Title of the Page
```

Choose a file name that doesn't collide with other files generated by the documentation tool. If the file name does not end in `.html`, it will be appended. The page title will appear at the top of the page as well as in the page index.

The lines after the `@page` line up to the next `@page` line or the end of the comment will be used as the page body. One does not need to add a title because the documentation tool automatically inserts the one specified in the `@page` directive.

Example:

```
//
// @page structure.html, Directory Structure
//
// The model core model files and the examples have been placed
// into different directories. The <tt>examples/</tt> directory...
//
//
// @page examples.html, Examples
// ...
//
```

One can create links to the generated pages using standard HTML, using the `<a href="...">...</a>` tag. All HTML files are placed in a single directory, so one doesn't have to worry about directories.

Example:

```
//
// @titlepage
// ...
// The structure of the model is described <a href="structure.html">here</a>.
//
```

14.5.3 Incorporating Externally Created Pages

The `@externalpage` directive allows one to add externally created pages into the generated documentation. `@externalpage` may appear in a file-level comment, and has a similar syntax as `@page`:

```
// @externalpage filename.html, Title of the Page
```

The directive causes the page to appear in the page index. However, the documentation tool does not check if the page exists, and it is the user's responsibility to copy the file into the directory of the generated documentation.

External pages can be linked to from other pages using the `<a href="...">...</a>` tag.

14.5.4 File Inclusion

The `@include` directive allows one to include the content of a file into a documentation comment. `@include` expects file name or path; if a relative path is given, it is interpreted as relative to the file that includes it.

The line of the `@include` directive will be replaced by the content of the file. The lines of the included file do not need to start with `//`, but otherwise they are processed in the same way as the NED comments. They can include other files, but circular includes are not allowed.

```
// ...
// @include ../copyright.txt
// ...
```

14.5.5 Extending Type Pages with Extra Content

Sometimes it is useful to customize the generated documentation pages that describe NED and MSG types by adding extra content. It is possible to provide a documentation fragment file in XML format which can be used by the documentation tool to add it to the generated documentation.

The fragment file may contain multiple top-level `<docfragment>` elements in the XML file's root element. Each `<docfragment>` element must have one of the `nedtype`, `msgtype` or `filename` attributes depending on which page it extends. Additionally, it must provide an `anchor` attribute to define a point in the page where the fragment's content should be inserted. The content of the fragment must be provided in a `<![CDATA[]]>` section.

```
<docfragments>
    <docfragment nedtype="fully.qualified.NEDTypeName" anchor="after-signals">
    <![CDATA[
```

```
        <h3 class="subtitle">Doc fragment after the signals section</h3>
        ...
    ]]>
    </docfragment>
    <docfragment msgtype="fully.qualified.MSGType" anchor="top">
    <![CDATA[
        <h3 class="subtitle">Doc fragment at the top of MSG type page</h3>
        ...
    ]]>
    </docfragment>
    <docfragment filename="project_relative_path/somefile.msg" anchor="bottom">
    <![CDATA[
        <h3 class="subtitle">Doc fragment at the end of the file listing page</h3>
        ...
    ]]>
    </docfragment>
</docfragments>
```

Possible attribute values:

- `nedtype`: Fully qualified NED type name.
- `msgtype`: Fully qualified MSG type name.
- `filename`: Project relative file path of a NED or MSG source file. The fragment will be inserted in the file's source listing page.
- `anchor`: Specifies the place where the content should inserted. Possible values for NED type: `top`, `after-types`, `after-description`, `after-image`, `after-diagrams`, `after-usage`, `after-inheritance`, `after-parameters`, `after-properties`, `after-gates`, `after-signals`, `after-statistics`, `after-unassigned-parameters`, `bottom`; for MSG type: `top`, `after-description`, `after-diagrams`, `after-inheritance`, `after-fields`, `after-properties`, `bottom`; for file listing: `top`, `after-types`, `bottom`

Chapter 15

Testing

15.1 Overview

15.1.1 Verification, Validation

Correctness of the simulation model is a primary concern of the developers and users of the model, because they want to obtain credible simulation results. Verification and validation are activities conducted during the development of a simulation model with the ultimate goal of producing an accurate and credible model.

- **Verification** of a model is the process of confirming that it is correctly implemented with respect to the conceptual model, that is, it matches specifications and assumptions deemed acceptable for the given purpose of application. During verification, the model is tested to find and fix errors in the implementation of the model.
- **Validation** checks the accuracy of the model's representation of the real system. Model validation is defined to mean "substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model". A model should be built for a specific purpose or set of objectives and its validity determined for that purpose.

Of the two, verification is essentially a software engineering issue, so it can be assisted with tools used for software quality assurance, for example testing tools. Validation is not a software engineering issue.

15.1.2 Unit Testing, Regression Testing

As mentioned above, software testing techniques can be of significant help during model verification. Testing can also help to ensure that a simulation model that once passed validation and verification will also remain correct for an extended period.

Software testing is an art on its own, with several techniques and methodologies. Here we'll only mention two types that are important for us, regression testing and unit testing.

- **Regression testing** is a technique that seeks to uncover new software bugs, or regressions, in existing areas of a system after changes such as enhancements, patches or configuration changes, have been made to them.

- **Unit testing** is a method by which individual units of source code are tested to determine if they are fit for use. In an object-oriented environment, this is usually done at the class level.

The two may overlap; for example, unit tests are also useful for discovering regressions.

One way of performing regression testing on an OMNEST simulation model is to record the log produced during simulation, and compare it to a pre-recorded log. The drawback is that code refactoring may nontrivially change the log as well, making it impossible to compare to the pre-recorded one. Alternatively, one may just compare the result files or only certain simulation results and be free of the refactoring effects, but then certain regressions may escape the testing. This type of tradeoff seems to be typical for regression testing.

Unit testing of simulation models may be done on class level or module level. There are many open-source unit testing frameworks for C++, for example CppUnit, Boost Test, Google Test, UnitTest++, just to name a few. They are well suited for class-level testing. However, they are usually cumbersome to apply to testing modules due to the peculiarities of the domain (network simulation) and OMNEST.

A test in an *xUnit*-type testing framework (a collective name for CppUnit-style frameworks) operates with various assertions to test function return values and object states. This approach is difficult to apply to the testing of OMNEST modules that often operate in a complex environment (cannot be easily instantiated and operated in isolation), react to various events (messages, packets, signals, etc.), and have complex dynamic behavior and substantial internal state.

Later sections will introduce `opp_test`, a tool OMNEST provides for assisting various testing task; and summarize various testing methods useful for testing simulation models.

15.2 The opp_test Tool

15.2.1 Introduction

This section documents the `opp_test`, a versatile tool that is helpful for various testing scenarios. `opp_test` can be used for various types of tests, including unit tests and regression tests. It was originally written for testing the OMNEST simulation kernel, but it is equally suited for testing functions, classes, modules, and whole simulations.

`opp_test` is built around a simple concept: it lets one define simulations in a concise way, runs them, and checks that the output (result files, log, etc.) matches a predefined pattern or patterns. In many cases, this approach works better than inserting various assertions into the code (which is still also an option).

Each test is a single file, with the `.test` file extension. All NED code, C++ code, ini files and other data necessary to run the test case as well as the PASS criteria are packed together in the test file. Such self-contained tests are easier to handle, and also encourage authors to write tests that are compact and to the point.

Let us see a small test file, `cMessage_properties_1.test`:

```
%description:
Test the name and length properties of cPacket.

%activity:
cPacket *pk = new cPacket();
```

```
pk->setName("ACK");
pk->setByteLength(64);
EV << "name: " << pk->getName() << endl;
EV << "length: " << pk->getByteLength() << endl;
delete pk;

%contains: stdout
name: ACK
length: 64
```

What this test says is this: create a simulation with a simple module that has the above C++ code block as the body of the `activity()` method, and when run, it should print the text after the `%contains` line.

To run this test, we need a *control script*, for example `runtest` from the `omnetpp/test/core` directory. `runtest` itself relies on the `opp_test` tool.

> **NOTE:** The control script is not part of OMNEST because it is somewhat specific to the simulation model or framework being tested, but it is usually trivial to write. A later section will explain how write the control script.

The output will be similar to this one:

```
$ ./runtest cMessage_properties_1.test
opp_test: extracting files from *.test files into work...
Creating Makefile in omnetpp/test/core/work...
cMessage_properties_1/test.cc
Creating executable: out/gcc-debug/work
opp_test: running tests using work.exe...
*** cMessage_properties_1.test: PASS
=======================================
PASS: 1   FAIL: 0   UNRESOLVED: 0

Results can be found in work/
```

This was a passing test. What would constitute a fail?

- crash
- simulation runtime error
- nonzero exit code (a simulation runtime error is also detected by nonzero exit code)
- the output doesn't match the expectation (there are several possibilities for expressing what is expected: multiple match criteria, literal string vs regex, positive vs negative match, matching against the standard output, standard error or any file, etc.)

One normally wants to run several tests together. The `runtest` script accepts several `.test` files on the command line, and when started without arguments, it defaults to `*.test`, all test files in the current directory. At the end of the run, the tool prints summary statistics (number of tests passed, failed, and being unresolved).

An example run from `omnetpp/test/core` (some lines were removed from the output, and one test was changed to show a failure):

```
$ ./runtest cSimpleModule-*.test
opp_test: extracting files from *.test files into work...
Creating Makefile in omnetpp/test/core/work...
[...]
Creating executable: out/gcc-debug/work
opp_test: running tests using work...
*** cSimpleModule_activity_1.test: PASS
*** cSimpleModule_activity_2.test: PASS
[...]
*** cSimpleModule_handleMessage_2.test: PASS
*** cSimpleModule_initialize_1.test: PASS
*** cSimpleModule_multistageinit_1.test: PASS
*** cSimpleModule_ownershiptransfer_1.test: PASS
*** cSimpleModule_recordScalar_1.test: PASS
*** cSimpleModule_recordScalar_2.test: FAIL (test-1.sca fails %contains-regex(2) r
expected pattern:
>>>>run General-1-.*?
scalar Test     one     24.2
scalar Test     two     -1.5888<<<<
actual output:
>>>>version 2
run General-1-20141020-11:39:34-1200
attr configname General
attr datetime 20141020-11:39:34
attr experiment General
attr inifile _defaults.ini
[...]
scalar Test     one     24.2
scalar Test     two     -1.5
<<<<
*** cSimpleModule_recordScalar_3.test: PASS
*** cSimpleModule_scheduleAt_notowner_1.test: PASS
*** cSimpleModule_scheduleAt_notowner_2.test: PASS
[...]
=======================================
PASS: 36   FAIL: 1   UNRESOLVED: 0
FAILED tests: cSimpleModule_recordScalar_2.test

Results can be found in work/
```

Note that code from all tests were linked to form a single executable, which saves time and disk space compared to per-test executables or libraries.

A test file like the one above is useful for unit testing of classes or functions. However, as we will see, the test framework provides further facilities that make it convenient for testing modules and whole simulations as well.

The following sections go into details about the syntax and features of `.test` files, about writing the control script, and give advice on how to cover several use cases with the `opp_test` tool.

15.2.2 Terminology

The next sections will use the following language:

- *test file*: A file with the `.test` extension that `opp_test` understands.
- *test tool*: The `opp_test` program
- *control script*: A script that relies on `opp_test` to run the tests. The control script is not part of OMNEST because it usually needs to be somewhat specific to the simulation model or framework being tested.
- *test program*: The simulation program whose output is checked by the test. It is usually `work/work` (`work/work.exe` on Windows). However, it is also possible to let the control script build a dynamic library from the test code, and then use e.g. `opp_run` as test program.
- *test directory*: The directory where a `.test` file is extracted; usually `work/<testname>/`. It is also set as working directory for running the test program.

15.2.3 Test File Syntax

Test files are composed of %-directives of the syntax:

```
%<directive>: <value>
<body>
```

The body extends up to the next directive (the next line starting with %), or to the end of the file. Some directives require a value, others a body, or both.

Certain directives, e.g. `%contains`, may occur several times in the file.

15.2.4 Test Description

Syntax:

```
%description:
<test-description-lines>
```

`%description` is customarily written at the top of the `.test` file, and lets one provide a multi-line comment about the purpose of the test. It is recommended to invest time into well-written descriptions, because they make determining the original purpose of a test that has become broken significantly easier.

15.2.5 Test Code Generation

This section describes the directives used for creating C++ source and other files in the test directory.

%activity

Syntax:

```
%activity:
<body-of-activity()>
```

`%activity` lets one write test code without much boilerplate. The directive generates a simple module that contains a single `activity()` method with the given code as method body.

A NED file containing the simple module's (barebones) declaration, and an ini file to set up the module as a network are also generated.

%module

Syntax:

```
%module: <modulename>
<simple-module-C++-definition>
```

`%module` lets one define a module class and run it as the only module in the simulation.

A NED file containing the simple module's (barebones) declaration, and an ini file to set up the module as a network are also generated.

%includes, %global

Syntax:

```
%includes:
<#include directives>
```

```
%global:
<global-code-pasted-before-activity>
```

`%includes` and `%global` are helpers for `%activity` and `%module`, and let one insert additional lines into the generated C++ code.

Both directives insert the code block above the module C++ declaration. The only difference is in their relation to the C++ namespace: the body of `%includes` is inserted above (i.e. outside) the namespace, and the body of `%globals` is inserted inside the namespace.

The Default Ini File

The following ini file is always generated:

```
[General]
network = <network-name>
cmdenv-express-mode = false
```

The network name in the file is chosen to match the module generated with `%activity` or `%module`; if they are absent, it will be `Test`.

%network

Syntax:

```
%network: <network-name>
```

This directive can be used to override the network name in the default ini file.

%file, %inifile

Syntax:

```
%file: <file-name>
<file-contents>
```

```
%inifile: [<inifile-name>]
<inifile-contents>
```

`%file` saves a file with the given file name and content into the test's extraction folder in the preparation phase of the test run. It is customarily used for creating NED files, MSG files, ini files, and extra data files required by the test. There can be several `%file` sections in the test file.

`%inifile` is similar to `%file` in that it also saves a file with the given file name and content, but it additionally also adds the file to the simulation's command line, causing the simulation to read it as an (extra) ini file. There can be several `%inifile` sections in the test file.

The default ini file is always generated.

The @TESTNAME@ Macro

In test files, the string `@TESTNAME@` will be replaced with the test case name. Since it is substituted everywhere (C++, NED, msg and ini files), one can also write things like `@TESTNAME@_function()`, or `printf("this is @TESTNAME@\n")`.

Avoiding C++ Name Clashes

Since all sources are compiled into a single test executable, actions have to be taken to prevent accidental name clashes between C++ symbols in different test cases. A good way to ensure this is place all code into namespaces named after the test cases.

```
namespace @TESTNAME@ {
    ...
};
```

This is done automatically for the `%activity`, `%module`, `%global` blocks, but for other files (e.g. source files generated via `%file`, that needs to be done manually.

15.2.6 PASS Criteria

%contains, %contains-regex, %not-contains, %not-contains-regex

Syntax:

```
%contains: <output-file-to-check>
<multi-line-text>

%contains-regex: <output-file-to-check>
<multi-line-regexp>

%not-contains: <output-file-to-check>
<multi-line-text>

%not-contains-regex: <output-file-to-check>
<multi-line-regexp>
```

These directives let one check for the presence (or absence) of certain text in the output. One can check a file, or the standard output or standard error of the test program; for the latter two, `stdout` and `stderr` needs to be specified as file name, respectively. If the file is not found, the test will be marked as *error*. There can be several `%contains`-style directives in the test file.

The text or regular expression can be multi-line. Before match is attempted, trailing spaces are removed from all lines in both the pattern and the file contents; leading and trailing blank lines in the patterns are removed; and any substitutions are performed (see `%subst`). Perl-style regular expressions are accepted.

To facilitate debugging of tests, the text/regex blocks are saved into the test directory.

%subst

Syntax:

```
%subst: /<search-regex>/<replacement>/<flags>
```

It is possible to apply text substitutions to the output before it is matched against expected output. This is done with `%subst` directive; there can be more than one `%subst` in a test file. It takes a Perl-style regular expression to search for, a replacement text, and flags, in the */search/replace/flags* syntax. Flags can be empty or a combination of the letters `i`, `m`, and `s`, for case-insensitive, multi-line or single-string match (see the Perl regex documentation.)

`%subst` was primarily invented to deal with differences in printf output across platforms and compilers: different compilers print infinite and not-a-number in different ways: `1.#INF`, `inf`, `Inf`, `-1.#IND`, `nan`, `NaN` etc. With `%subst`, they can be brought to a common form:

```
%subst: /-?1\.#INF/inf/
%subst: /-?1\.#IND/nan/
%subst: /-?1\.#QNAN/nan/
%subst: /-?NaN/nan/
%subst: /-?nan/nan/
```

%exitcode, %ignore-exitcode

Syntax:

```
%exitcode: <one-or-more-numeric-exit-codes>

%ignore-exitcode: 1
```

%exitcode and %ignore-exitcode let one test the exit code of the test program. The former checks that the exit code is one of the numbers specified in the directive; the other makes the test framework ignore the exit code.

OMNEST simulations exit with zero if the simulation terminated without an error, and some >0 code if a runtime error occurred. Normally, a nonzero exit code makes the test fail. However, if the expected outcome is a runtime error (e.g. for some negative test cases), one can use either %exitcode to express that, or specify %ignore-exitcode and test for the presence of the correct error message in the output.

%file-exists, %file-not-exists

Syntax:

```
%file-exists: <filename>
```

```
%file-not-exists: <filename>
```

These directives test for the presence or absence of a certain file in the test directory.

15.2.7 Extra Processing Steps

%env, %extraargs, %testprog

Syntax:

```
%env: <environment-variable-name>=<value>
```

```
%extraargs: <argument-list>
```

```
%testprog: <executable>
```

The %env directive lets one set an environment variable that will be defined when the test program and the potential pre- and post-processing commands run. There can be multiple %env directives in the test file.

%extraargs lets one add extra command-line arguments to the test program (usually the simulation) when it is run.

The %testprog directive lets one replace the test program. %testprog also slightly alters the arguments the test program is run with. Normally, the test program is launched with the following command line:

```
$ <default-testprog> -u Cmdenv <test-extraargs> <global-extraargs> <inifiles>
```

When %testprog is present, it becomes the following:

```
$ <custom-testprog> <test-extraargs> <global-extraargs>
```

That is, -u Cmdenv and <inifilenames> are removed; this allows one to invoke programs that do not require or understand them, and puts the test author in complete command of the arguments list.

Note that %extraargs and %testprog have an equivalent command-line option in opp_test. (In the text above, <global-extraargs> stands for extra args specified to opp_test.) %env

doesn't need an option in `opp_test`, because the test program inherits the environment variables from `opp_test`, so one can just set them in the control script, or in the shell one runs the tests from.

%prerun-command, %postrun-command

Syntax:

```
%prerun-command: <command>
```

```
%postrun-command: <command>
```

These directives let one run extra commands before/after running the test program (i.e. the simulation). There can be multiple pre- and post-run commands. The post-run command is useful when the test outcome cannot be determined by simple text matching, but requires statistical evaluation or other processing.

If the command returns a nonzero exit code, the test framework will assume that it is due to a technical problem (as opposed to test failure), and count the test as *error*. To make the test fail, let the command write a file, and match the file's contents using `%contains` & co.

If the post-processing command is a short script, it is practical to add it into the `.test` file via the `%file` directive, and invoke it via its interpreter. For example:

```
%postrun-command: python test.py
%file: test.py
<Python script>
```

Or:

```
%postrun-command: R CMD BATCH test.R
%file: test.R
<R script>
```

If the script is very large or shared among several tests, it is more practical to place it into a separate file. The test command can find the script e.g. by relative path, or by referring to an environment variable that contains its location or full path.

15.2.8 Error

A test case is considered to be in *error* if the test program cannot be executed at all, the output cannot be read, or some other technical problem occurred.

15.2.9 Expected Failure

`%expected-failure` can be used in the test file to force a test case to ignore a failure. If a test case marked with `%expected-failure` fails, it will be counted as *expectfail* instead of *fail*. `opp_test` will return successfully if no test cases reported *fail* or *error* results.

Syntax:

```
%expected-failure: <single-line-reason-for-allowing-a-failure>
```

15.2.10 Skipped

A test case can be skipped if the current system configuration does not allow its execution (e.g. certain optional features are not present). Skipping is done by printing `#SKIPPED` or `#SKIPPED:some-explanation` on the standard output, at the beginning of the line.

15.2.11 opp_test Synopsys

Little has been said so far what `opp_test` actually does, or how it is meant to be run. `opp_test` has two modes: file generation and test running. When running a test suite, `opp_test` is actually run twice, once in file generation mode, then in test running mode.

File generation mode has the syntax `opp_test gen <options> <testfiles>`. For example:

```
$ opp_test gen *.test
```

This command will extract C++ and NED files, ini files, etc., from the `.test` files into separate files. All files will be created in a work directory (which defaults to `./work/`), and each test will have its own subdirectory under `./work/`.

The second mode, test running, is invoked as `opp_test run <options> <testfiles>`. For example:

```
$ opp_test run *.test
```

In this mode, `opp_test` will run the simulations, check the results, and report the number of passes and failures. The way of invoking simulations (which executable to run, the list of command-line arguments to pass, etc.) can be specified to `opp_test` via command-line options.

> **NOTE:** Run `opp_test` in your OMNEST installation to get the exact list of command-line options.

The simulation needs to have been built from source before `opp_test run` can be issued. Usually one would employ a command similar to

```
$ cd work; opp_makemake --deep; make
```

to achieve that.

15.2.12 Writing the Control Script

Usually one writes a control script to automate the two invocations of `opp_test` and the build of the simulation model between them.

A basic variant would look like this:

```
#! /bin/sh
opp_test gen -v *.test || exit 1
(cd work; opp_makemake -f --deep; make) || exit 1
opp_test run -v *.test
```

For any practical use, the test suite needs to refer to the codebase being tested. This means that the codebase must be added to the include path, must be linked with, and the NED

files must be added to the NED path. The first two can be achieved by the appropriate parameterization of `opp_makemake`; the last one can be done by setting and exporting the `NEDPATH` environment variable in the control script.

For inspiration, check out `runtest` in the `omnetpp/test/core` directory, and a similar script used in the INET Framework.

* * *

Further sections describe how one can implement various types of tests in OMNEST.

15.3 Smoke Tests

Smoke tests are a tool for very basic verification and regression testing. Basically, the simulation is run for a while, and it must not crash or stop with a runtime error. Naturally, smoke test provide very low confidence in the model, but in turn they are very easy to implement.

Automation is important. The INET Framework contains a script that runs all or selected simulations defined in a CSV file (with columns like the working directory and the command to run), and reports the results. The script can be easily adapted to other models or model frameworks.

15.4 Fingerprint Tests

Fingerprint tests are a low-cost but effective tool for regression testing of simulation models. A fingerprint is a hash computed from various properties of simulation events, messages and statistics. The hash value is continuously updated as the simulation executes, and thus, the final fingerprint value is a characteristic of the simulation's trajectory. For regression testing, one needs to compare the computed fingerprints to that from a reference run – if they differ, the simulation trajectory has changed. In general, fingerprint tests are very useful for ensuring that a change (some refactoring, a bugfix, or a new feature) didn't break the simulation.

15.4.1 Fingerprint Computation

Technically, providing a **`fingerprint`** option in the config file or on the command line (`--fingerprint=...`) will turn on fingerprint computation in the OMNEST simulation kernel. When the simulation terminates, OMNEST compares the computed fingerprints with the provided ones, and if they differ, an error is generated.

Ingredients

The fingerprint computation algorithm allows controlling what is included in the hash value. Changing the *ingredients* allows one to make the fingerprint sensitive for certain changes while keeping it immune to others.

The ingredients of a fingerprint are usually indicated after a / sign following the hexadecimal hash value. Each ingredient is identified with a letter. For example, **t** stands for simulation time. Thus, the following `omnetpp.ini` line

```
fingerprint = 53de-64a7/tplx
```

means that a fingerprint needs to be computed with the simulation time, the module full path, received packet's bit length and the extra data included for each event, and the result should be `53de-64a7`.

The full list of fingerprint ingredients:

- **e** : event number
- **t** : simulation time
- **n** : message/event full name
- **c** : message/event class name
- **k** : message kind
- **l** : message (packet) bit length
- **o** : message control info class name
- **d** : message data
- **i** : module id
- **m** : module full name (name with index)
- **p** : module full path (hierarchical name)
- **a** : module class name
- **r** : random numbers drawn
- **s** : scalar results
- **z** : statistic results (histogram, etc.)
- **v** : vector results
- **x** : extra data added programmatically

Ingredients may also be specified with the **`fingerprint-ingredients`** configuration option. However, that is rarely necessary, because the ingredients list included in the fingerprints take precedence, and are also more convenient to use.

Multiple Fingerprints, Alternative Values

It is possible to specify more than one fingerprint, separated by *commas*, each with different ingredients. This will cause OMNEST to compute multiple fingerprints, and all of them must match for the test to pass. An example:

```
fingerprint = 53de-64a7/tplx, 9a3f-7ed2/szv
```

Occasionally, the same simulation gives a different fingerprint when run on a different processor architecture or platform. This is due to subtle differences in floating point arithmetic across platforms.[1] Acknowledging this fact, OMNEST lets one list several values for a fingerprint, separated by *spaces*, and will accept whichever is produced by the simulation. The following example lists two alternative values for both fingerprints.

```
fingerprint = 53de-64a7/tplx 63dc-ff21/tplx, 9a3f-7ed2/szv da39-91fc/szv
```

Note that fingerprint computation has been changed and significantly extended in OMNEST version 5.0.[2]

[1] There are differences between the floating point operations of AMD and Intel CPUs. Running under a processor emulator like `valgrind` may also produce a different fingerprint. This is normal. Hint: see gcc options `-mfpmath=sse -msse2`.

[2] The old (OMNEST 4.x) fingerprint was computed from the module ID and simulation time of each event. To reproduce a 4.x fingerprint on OMNEST 5.0 or later, compile OMNEST and the model with `USE_OMNETPP4x_FINGERPRINTS` defined. Simply setting the ingredients to **ti** is not enough because of additional, subtle changes in the simulation kernel.

Further Filtering

It is also possible to filter which modules, statistics, etc. are included in the fingerprints. The **fingerprint-events**, **fingerprint-modules**, and **fingerprint-results** options filter by events, modules, and statistical results, respectively. These options take wildcard expressions that are matched against the corresponding object before including its property in the fingerprint. These filters are mainly useful to limit fingerprint computation to certain parts of the simulation.

Programmatic Access

`cFingerprintCalculator` is the class responsible for fingerprint computation. The current fingerprint computation object can be retrieved from `cSimulation`, using the `getFingerprintCalculator()` member function. This method will return `nullptr` if fingerprint computation is turned off for the current simulation run.

To contribute data to the fingerprint, `cFingerprintCalculator` has several `addExtraData()` methods for various data types (string, `long`, `double`, byte array, etc.)

An example (note that we check the pointer for `nullptr` to decide whether a fingerprint is being computed):

```
cFingerprintCalculator *fingerprint = getSimulation()->getFingerprintCalculator();
if (fingerprint) {
    fingerprint->addExtraData(retryCount);
    fingerprint->addExtraData(rttEstimate);
}
```

Data added using `addExtraData()` will only be counted in the fingerprint if the list of fingerprint ingredients contains **x** (otherwise `addExtraData()` does nothing).

15.4.2 Fingerprint Tests

The INET Framework contains a script for automated fingerprint tests as well. The script runs all or selected simulations defined in a CSV file (with columns like the working directory, the command to run, the simulation time limit, and the expected fingerprints), and reports the results. The script is extensively used during INET Framework development to detect regressions, and can be easily adapted to other models or model frameworks.

Exerpt from a CSV file that prescribes fingerprint tests to run:

```
examples/aodv/, ./run -f omnetpp.ini -c Static,   50s,  4c29-95ef/tplx
examples/aodv/, ./run -f omnetpp.ini -c Dynamic,  60s,  8915-f239/tplx
examples/dhcp/, ./run -f omnetpp.ini -c Wired,    800s, e88f-fee0/tplx
examples/dhcp/, ./run -f omnetpp.ini -c Wireless, 500s, faa5-4111/tplx
```

15.5 Unit Tests

If a simulation models contains units of code (classes, functions) smaller than a module, they are candidates for unit testing. For a network simulation model, examples of such classes are network addresses, fragmentation reassembly buffers, queues, various caches and tables, serializers and deserializers, checksum computation, etc.

Unit tests can be implemented as `.test` files using the `opp_test` tool (the `%activity` directive is especially useful here), or with potentially any other C++ unit testing framework.

When using `.test` files, the *build* part of the control script needs to be set up so that it adds the tested library's source folder(s) to the include path, and also links the library to the test code.

15.6 Module Tests

OMNEST modules are not as easy to unit test as standalone classes, because they typically assume a more complex environment, and, especially modules that implement network protocols, participate in more complex interactions than the latter.

To test a module in isolation, one needs to place it into a simulation where the module's normal operation environment (i.e. other modules it normally communicates with) are replaced by mock objects. Mock objects are responsible for providing stimuli for the module under test, and (partly) for checking the response.

Module tests may be implemented in `.test` files using the `opp_test` tool. A `.test` file allows one to place the test description, the test setup and the expected output into a single, compact file, while large files or files shared among several tests may be factored out and only referenced by `.test` files.

15.7 Statistical Tests

Statistical tests are those where the test outcome is decided on some statistical property or properties of the simulation results.

Statistical tests may be useful as validation as well as regression testing.

15.7.1 Validation Tests

Validation tests aim to verify that simulation results correspond to some reference values, ideally to those obtained from the real system. In practice, reference values may come from physical measurements, theoretical values, or another simulator's results.

15.7.2 Statistical Regression Tests

After a refactoring that changes the simulation trajectory (e.g. after eliminating or introducing extra events, or changes in RNG usage), there may be no other way to do regression testing than checking that the model produces *statistically* the same results as before.

For statististical regression tests, one needs to perform several simulation runs with the same configuration but different RNG seeds, and verify that the results are from the same distributions as before. One can use *Student's t-test* (for mean) and the *F-test* (for variance) to check that the "before" and the "after" sets of results are from the same distribution.

15.7.3 Implementation

Statistical software like *GNU R* is extremely useful for these tests.

Statistical tests may also be implemented in `.test` files. To let the tool run several simulations within one test, one may use `%extraargs` to pass the `-r <runs>` option to Cmdenv; alternatively, one may use `%testprog` to have the test tool run `opp_runall` instead of the normal simulation program. For doing the statistical computations, one may use `%postrun-command` to run an R script. The R script may rely on the `omnetpp` R package for reading the result files.

The INET Framework contains statistical tests where one can look for inspiration.

Chapter 16

Parallel Distributed Simulation

16.1 Introduction to Parallel Discrete Event Simulation

OMNEST supports parallel execution of large simulations. This section provides a brief picture of the problems and methods of parallel discrete event simulation (PDES). Interested readers are strongly encouraged to look into the literature.

For parallel execution, the model is to be partitioned into several LPs (logical processes) that will be simulated independently on different hosts or processors. Each LP will have its own local Future Event Set, and thus will maintain its own local simulation time. The main issue with parallel simulations is keeping LPs synchronized in order to avoid violating the causality of events. Without synchronization, a message sent by one LP could arrive in another LP when the simulation time in the receiving LP has already passed the timestamp (arrival time) of the message. This would break causality of events in the receiving LP.

There are two broad categories of parallel simulation algorithms that differ in the way they handle causality problems outlined above:

1. **Conservative algorithms** prevents incausalities from happening. The Null Message Algorithm exploits knowledge of the time when LPs send messages to other LPs, and uses special *null messages* to propagate this information to other LPs. If an LP knows it won't receive any messages from other LPs until $t + \Delta t$ simulation time, it may advance until $t + \Delta t$ without the need for external synchronization. Conservative simulation tends to converge to sequential simulation (slowed down by communication between LPs) if there is not enough parallelism in the model, or parallelism is not exploited by sending a sufficient number of null messages.

2. **Optimistic synchronization** allows incausalities to occur, but detects and repairs them. Repairing involves rollbacks to a previous state, sending out anti-messages to cancel messages sent out during the period that is being rolled back, etc. Optimistic synchronization is extremely difficult to implement, because it requires periodic state saving and the ability to restore previous states. In any case, implementing optimistic synchronization in OMNEST would require – in addition to a more complicated simulation kernel – writing significantly more complex simple module code from the user. Optimistic synchronization may be slow in cases of excessive rollbacks.

16.2 Assessing Available Parallelism in a Simulation Model

OMNEST currently supports conservative synchronization via the classic Chandy-Misra-Bryant (or null message) algorithm [CM79]. To assess how efficiently a simulation can be parallelized with this algorithm, we'll need the following variables:

- P *performance* represents the number of events processed per second (ev/sec). [1] P depends on the performance of the hardware and the computation-intensiveness of processing an event. P is independent of the size of the model. Depending on the nature of the simulation model and the performance of the computer, P is usually in the range of 20,000..500,000 ev/sec.

- E *event density* is the number of events that occur per simulated second (ev/simsec). E depends on the model only, and not where the model is executed. E is determined by the size, the detail level and also the nature of the simulated system (e.g. cell-level ATM models produce higher E values than call center simulations.)

- R *relative speed* measures the simulation time advancement per second (simsec/sec). R strongly depends on both the model and on the software/hardware environment where the model executes. Note that $R = P/E$.

- L *lookahead* is measured in simulated seconds (simsec). When simulating telecommunication networks and using link delays as lookahead, L is typically in the msimsec-μsimsec range.

- τ *latency* (sec) characterizes the parallel simulation hardware. τ is the latency of sending a message from one LP to another. τ can be determined using simple benchmark programs. The authors' measurements on a Linux cluster interconnected via a 100Mb Ethernet switch using MPI yielded τ=22μs which is consistent with measurements reported in [OF00]. Specialized hardware such as Quadrics Interconnect [Qua] can provide τ=5μs or better.

In large simulation models, P, E and R usually stay relatively constant (that is, display little fluctuations in time). They are also intuitive and easy to measure. The OMNEST displays these values on the GUI while the simulation is running, see Figure 16.1. Cmdenv can also be configured to display these values.

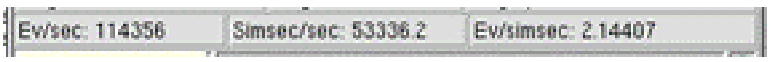


Figure 16.1: Performance bar in OMNEST showing P, R and E

After having approximate values of P, E, L and τ, calculate the λ *coupling factor* as the ratio of LE and τP:

$\lambda = (LE)/(\tau P)$

Without going into the details: if the resulting λ value is at minimum larger than one, but rather in the range 10..100, there is a good chance that the simulation will perform well when run in parallel. With $\lambda < 1$, poor performance is guaranteed. For details see the paper [VŞE03].

[1] Notations: *ev:* events, *sec:* real seconds, *simsec:* simulated seconds

16.3 Parallel Distributed Simulation Support in OMNEST

16.3.1 Overview

This chapter presents the parallel simulation architecture of OMNEST. The design allows simulation models to be run in parallel without code modification – it only requires configuration. The implementation relies on the approach of placeholder modules and proxy gates to instantiate the model on different LPs – the placeholder approach allows simulation techniques such as topology discovery and direct message sending to work unmodified with PDES. The architecture is modular and extensible, so it can serve as a framework for research on parallel simulation.

The OMNEST design places a big emphasis on *separation of models from experiments*. The main rationale is that usually a large number of simulation experiments need to be done on a single model before a conclusion can be drawn about the real system. Experiments tend to be ad-hoc and change much faster than simulation models; thus it is a natural requirement to be able to carry out experiments without disturbing the simulation model itself.

Following the above principle, OMNEST allows simulation models to be executed in parallel without modification. No special instrumentation of the source code or the topology description is needed, as partitioning and other PDES configuration is entirely described in the configuration files.

OMNEST supports the Null Message Algorithm with static topologies, using link delays as lookahead. The laziness of null message sending can be tuned. Also supported is the Ideal Simulation Protocol (ISP) introduced by Bagrodia in 2000 [BT00]. ISP is a powerful research vehicle to measure the efficiency of PDES algorithms, both optimistic and conservative; more precisely, it helps determine the maximum speedup achievable by any PDES algorithm for a particular model and simulation environment. In OMNEST, ISP can be used for benchmarking the performance of the Null Message Algorithm. Additionally, models can be executed without any synchronization, which can be useful for educational purposes (to demonstrate the need for synchronization) or for simple testing.

For the communication between LPs (logical processes), OMNEST primarily uses MPI, the Message Passing Interface standard [For94]. An alternative communication mechanism is based on named pipes, for use on shared memory multiprocessors without the need to install MPI. Additionally, a file system based communication mechanism is also available. It communicates via text files created in a shared directory, and can be useful for educational purposes (to analyse or demonstrate messaging in PDES algorithms) or to debug PDES algorithms. Implementation of a shared memory-based communication mechanism is also planned for the future, to fully exploit the power of multiprocessors without the overhead of and the need to install MPI.

Nearly every model can be run in parallel. The constraints are the following:

- modules may communicate via sending messages only (no direct method call or member access) unless mapped to the same processor
- no global variables
- there are some limitations on direct sending (no sending to a *sub*module of another module, unless mapped to the same processor)
- lookahead must be present in the form of link delays
- currently static topologies are supported (we are working on a research project that aims to eliminate this limitation)

PDES support in OMNEST follows a modular and extensible architecture. New communication mechanisms can be added by implementing a compact API (expressed as a C++ class) and registering the implementation – after that, the new communications mechanism can be selected for use in the configuration.

New PDES synchronization algorithms can be added in a similar way. PDES algorithms are also represented by C++ classes that have to implement a very small API to integrate with the simulation kernel. Setting up the model on various LPs as well as relaying model messages across LPs is already taken care of and not something the implementation of the synchronization algorithm needs to worry about (although it can intervene if needed, because the necessary hooks are provided).

The implementation of the Null Message Algorithm is also modular in itself in that the lookahead discovery can be plugged in via a defined API. Currently implemented lookahead discovery uses link delays, but it is possible to implement more sophisticated approaches and select them in the configuration.

16.3.2 Parallel Simulation Example

We will use the Parallel CQN example simulation for demonstrating the PDES capabilities of OMNEST. The model consists of N tandem queues where each tandem consists of a switch and k single-server queues with exponential service times (Figure 16.2). The last queues are looped back to their switches. Each switch randomly chooses the first queue of one of the tandems as destination, using uniform distribution. The queues and switches are connected with links that have nonzero propagation delays. Our OMNEST model for CQN wraps tandems into compound modules.

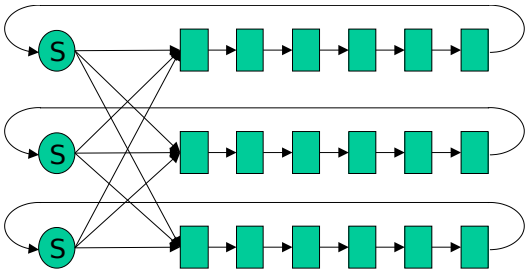


Figure 16.2: The Closed Queueing Network (CQN) model

To run the model in parallel, we assign tandems to different LPs (Figure 16.3). Lookahead is provided by delays on the marked links.

To run the CQN model in parallel, we have to configure it for parallel execution. In OMNEST, the configuration is in the `omnetpp.ini` file. For configuration, first we have to specify partitioning, that is, assign modules to processors. This is done by the following lines:

```
[General]
*.tandemQueue[0]**.partition-id = 0
*.tandemQueue[1]**.partition-id = 1
*.tandemQueue[2]**.partition-id = 2
```

The numbers after the equal sign identify the LP.

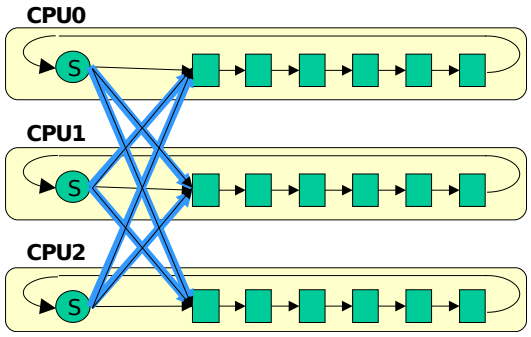


Figure 16.3: Partitioning the CQN model

Then we have to select the communication library and the parallel simulation algorithm, and enable parallel simulation:

```
[General]
parallel-simulation = true
parsim-communications-class = "cMPICommunications"
parsim-synchronization-class = "cNullMessageProtocol"
```

When the parallel simulation is run, LPs are represented by multiple running instances of the same program. When using LAM-MPI [LAM], the mpirun program (part of LAM-MPI) is used to launch the program on the desired processors. When named pipes or file communications is selected, the opp_prun OMNEST utility can be used to start the processes. Alternatively, one can run the processes by hand (the -p flag tells OMNEST the index of the given LP and the total number of LPs):

```
./cqn -p0,3 &
./cqn -p1,3 &
./cqn -p2,3 &
```

For PDES, one will usually want to select the command-line user interface, and redirect the output to files. (OMNEST provides the necessary configuration options.)

The graphical user interface of OMNEST can also be used (as evidenced by Figure 16.4), independently of the selected communication mechanism. The GUI interface can be useful for educational or demonstration purposes. OMNEST displays debugging output about the Null Message Algorithm, EITs and EOTs can be inspected, etc.

16.3.3 Placeholder Modules, Proxy Gates

When setting up a model partitioned to several LPs, OMNEST uses placeholder modules and proxy gates. In the local LP, placeholders represent sibling submodules that are instantiated on other LPs. With placeholder modules, every module has all of its siblings present in the local LP – either as placeholder or as the "real thing". Proxy gates take care of forwarding messages to the LP where the module is instantiated (see Figure 16.5).

The main advantage of using placeholders is that algorithms such as topology discovery embedded in the model can be used with PDES unmodified. Also, modules can use direct message sending to any sibling module, including placeholders. This is so because the destination

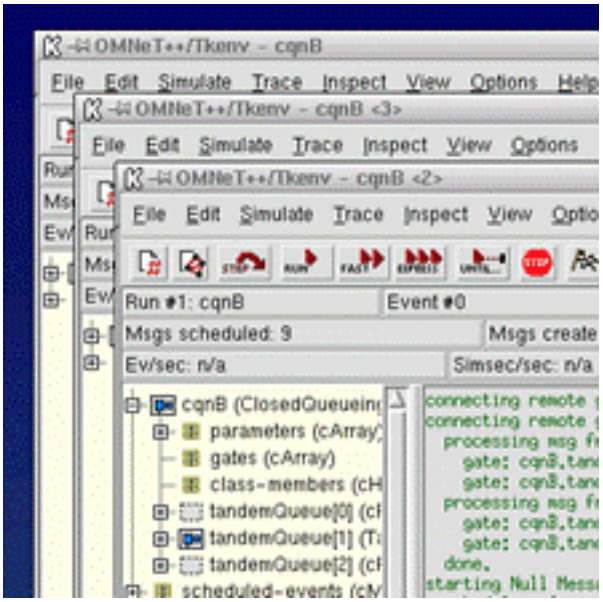


Figure 16.4: Screenshot of CQN running in three LPs

of direct message sending is an input gate of the destination module – if the destination module is a placeholder, the input gate will be a proxy gate which transparently forwards the messages to the LP where the "real" module was instantiated. A limitation is that the destination of direct message sending cannot be a *submodule* of a sibling (which is probably a bad practice anyway, as it violates encapsulation), simply because placeholders are empty and so its submodules are not present in the local LP.

Instantiation of compound modules is slightly more complicated. Since submodules can be on different LPs, the compound module may not be "fully present" on any given LP, and it may have to be present on several LPs (wherever it has submodules instantiated). Thus, compound modules are instantiated wherever they have at least one submodule instantiated, and are represented by placeholders everywhere else (Figure 16.6).

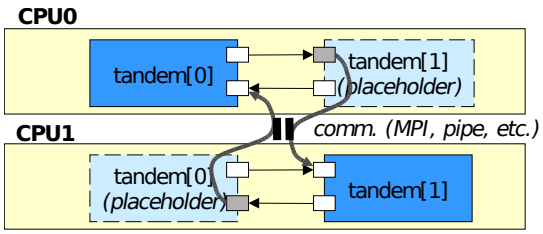


Figure 16.5: Placeholder modules and proxy gates

16.3.4 Configuration

Parallel simulation configuration is the `[General]` section of `omnetpp.ini`.

The parallel distributed simulation feature can be turned on with the **`parallel-simulation`**

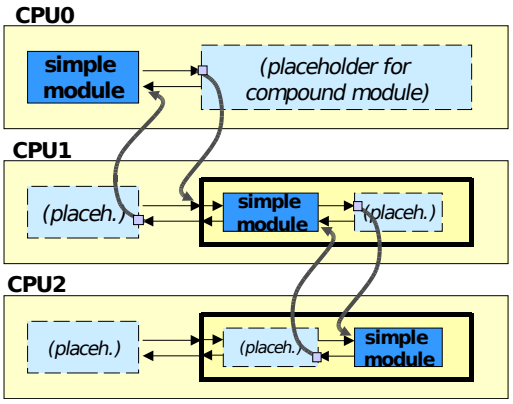


Figure 16.6: Instantiating compound modules

boolean option.

The **parsim-communications-class** selects the class that implements communication between partitions. The class must implement the `cParsimCommunications` interface.

The **parsim-synchronization-class** selects the parallel simulation algorithm. The class must implement the `cParsimSynchronizer` interface.

The following two options configure the Null Message Algorithm, so they are only effective if `cNullMessageProtocol` has been selected as synchronization class:

- **parsim-nullmessageprotocol-lookahead-class** selects the lookahead class for the NMA; the class must be subclassed from `cNMPLookahead`. The default class is `cLinkDelayLookahead`.
- **parsim-nullmessageprotocol-laziness** expects a number in the $(0,1)$ interval (the default is 0.5), and it ontrols how often NMA should send out null messages; the value is understood in proportion to the lookahead, e.g. 0.5 means every $lookahead/2$ simsec.

The **parsim-debug** boolean option enables/disables printing log messages about the parallel simulation algorithm. It is turned on by default, but for production runs we recommend turning it off.

Other configuration options configure MPI buffer sizes and other details; see options that begin with `parsim-` in Appendix I.

When you are using cross-mounted home directories (the simulation's directory is on a disk mounted on all nodes of the cluster), a useful configuration setting is

```
[General]
fname-append-host = true
```

It will cause the host names to be appended to the names of all output vector files, so that partitions don't overwrite each other's output files. (See section 11.20.3)

16.3.5 Design of PDES Support in OMNEST

The design of PDES support in OMNEST follows a layered approach, with a modular and extensible architecture. The overall architecture is depicted in Figure 16.7.

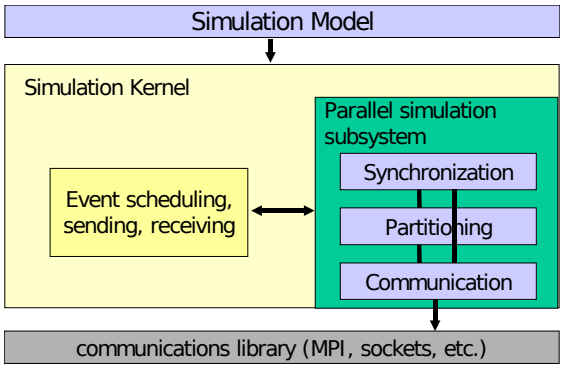


Figure 16.7: Architecture of OMNEST PDES implementation

The parallel simulation subsytem is an optional component itself, which can be removed from the simulation kernel if not needed. It consists of three layers, from the bottom up: Communications Layer, Partitioning Layer and Synchronization Layer.

The Communications Layer

The purpose of the Communications Layer is to provide elementary messaging services between partitions for the upper layer. The services include send, blocking receive, nonblocking receive and broadcast. The send/receive operations work with *buffers*, which encapsulate packing and unpacking operations for primitive C++ types. The message class and other classes in the simulation library can pack and unpack themselves into such buffers. The Communications layer API is defined in the `cParsimCommunications` interface (abstract class); specific implementations like the MPI one (`cMPICommunications`) subclass from this, and encapsulate MPI send/receive calls. The matching buffer class `cMPICommBuffer` encapsulates MPI pack/unpack operations.

The Partitioning Layer

The Partitioning Layer is responsible for instantiating modules on different LPs according to the partitioning specified in the configuration, for configuring proxy gates. During the simulation, this layer also ensures that cross-partition simulation messages reach their destinations. It intercepts messages that arrive at proxy gates and transmits them to the destination LP using the services of the Communications Layer. The receiving LP unpacks the message and injects it at the gate the proxy gate points at. The implementation basically encapsulates the `cParsimSegment`, `cPlaceholderModule`, `cProxyGate` classes.

The Synchronization Layer

The Synchronization Layer encapsulates the parallel simulation algorithm. Parallel simulation algorithms are also represented by classes, subclassed from the `cParsimSynchronizer` abstract class. The parallel simulation algorithm is invoked on the following hooks: event scheduling, processing model messages outgoing from the LP, and messages (model messages or internal messages) arriving from other LPs. The first hook, event scheduling, is a function

invoked by the simulation kernel to determine the next simulation event; it also has full access to the future event set (FES) and can add/remove events for its own use. Conservative parallel simulation algorithms will use this hook to block the simulation if the next event is unsafe, e.g. the null message algorithm implementation (`cNullMessageProtocol`) blocks the simulation if an EIT has been reached until a null message arrives (see [BT00] for terminology); also it uses this hook to periodically send null messages. The second hook is invoked when a model message is sent to another LP; the null message algorithm uses this hook to piggyback null messages on outgoing model messages. The third hook is invoked when any message arrives from other LPs, and it allows the parallel simulation algorithm to process its own internal messages from other partitions; the null message algorithm processes incoming null messages here.

The Null Message Protocol implementation itself is modular; it employs a separate, configurable lookahead discovery object. Currently only link delay based lookahead discovery has been implemented, but it is possible to implement more sophisticated types.

The Ideal Simulation Protocol (ISP; see [BT00]) implementation consists of two parallel simulation protocol implementations: the first one is based on the null message algorithm and additionally records the external events (events received from other LPs) to a trace file; the second one executes the simulation using the trace file to find out which events are safe and which are not.

Note that although we implemented a conservative protocol, the provided API itself would allow implementing optimistic protocols, too. The parallel simulation algorithm has access to the executing simulation model, so it could perform saving/restoring model state if model objects support this [2].

We also expect that because of the modularity, extensibility and clean internal architecture of the parallel simulation subsystem, the OMNEST framework has the potential to become a preferred platform for PDES research.

[2] Unfortunately, support for state saving/restoration needs to be individually and manually added to each class in the simulation, including user-programmed simple modules.

Chapter 17

Customizing and Extending OMNEST

17.1 Overview

OMNEST is an open system, and several details of its operation can be customized and extended by writing C++ code. Some extension interfaces have already been covered in other chapters:

- *Defining new NED functions* was described in 7.12
- *Defining new result filters and recorders* was described in 4.15.6

This chapter will begin by introducing some infrastructure features that are useful for extensions:

- *Config options*. This facility lets other extensions classes define their own configuration options.
- *Simulation lifecycle listeners* allow extensions to get notified when a network is set up, simulation is started, paused or resumed, the simulation ended successfully or with an error, and so on.
- `cEvent` lets extensions schedule actions for certain simulation times. This is especially useful for custom event schedulers that we'll cover later in this chapter.

Then we will continue with the descriptions of the following extension interfaces:

- `cRNG` lets one add new random number generator algorithms.
- `cScheduler` is an interface for event schedulers. This extension interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation.
- `cFutureEventSet`. This extension interface allows one to replace the data structure used for storing future events during simulation, i.e. the FES. This may make sense for specialized workloads.

- `cFingerprintCalculator`. This extension interface allows one to replace or extend the fingerprint computation algorithm.
- `cIOutputScalarManager`. This extension interface allows one to create additional means of saving scalar results, for example database or CSV output.
- `cIOutputVectorManager`. This extension interface allows one to create additional means of saving vector results, for example database or CSV output.
- `cIEventlogManager`. This extension interface allows one to customize event log recording.
- `cISnapshotManager`. It provides an output stream to which snapshots are written.
- `cConfigurationEx`. Configuration provider extension. This extension interface lets one replace `omnetpp.ini` with some other implementation, for example a database.
- *User interfaces*. When existing runtime user interfaces (Cmdenv, Qtenv) don't suffice, one can create a new one, reusing the infrastructure provided by the common base of the three.

Many extension interfaces follow a common pattern: one needs to implement a given interface class (e.g. `cRNG` for random number generators), let OMNEST know about it by registering the class with the `Register_Class()` macro, and finally activate it by the appropriate configuration option (e.g. `rng-class=MyRNG`). The interface classes (`cRNG`, `cScheduler`, etc.) are documented in the API Reference.

> **NOTE:** A common error is that OMNEST cannot find the class at runtime. When that happens, make sure the executable actually contains the code of the class. When linking with a library, over-optimizing linkers (esp. on Unix) tend to leave out code which seems to be unreferenced by other parts of the program.

The following sections elaborate on the various extension interfaces.

17.2 Adding a New Configuration Option

17.2.1 Registration

New configuration options need to be declared with one of the appropriate registration macros. These macros are:

```
Register_GlobalConfigOption(ID, NAME, TYPE, DEFAULTVALUE, DESCRIPTION)
Register_PerRunConfigOption(ID, NAME, TYPE, DEFAULTVALUE, DESCRIPTION)
Register_GlobalConfigOptionU(ID, NAME, UNIT, DEFAULTVALUE, DESCRIPTION)
Register_PerRunConfigOptionU(ID, NAME, UNIT, DEFAULTVALUE, DESCRIPTION)
Register_PerObjectConfigOption(ID, NAME, KIND, TYPE, DEFAULTVALUE, DESCRIPTION)
Register_PerObjectConfigOptionU(ID, NAME, KIND, UNIT, DEFAULTVALUE, DESCRIPTION)
```

Config options come in three flavors, as indicated by the macro names:

- *Global* options affect all configurations (i.e. they are only accepted in the `[General]` section but not in `[Config <name>]` sections)

- *Per-Run* options can be specified in any section (i.e. both in `[General]` and in `[Config <name>]` sections). They affect the configuration they occur in.
- *Per-Object* options can be specified in any section (i.e. both in `[General]` and in `[Config <name>]` sections). They are specific to an object or group of objects. Their names must always contain a hyphen (-) character so that they can be distinguished from module/channel parameter assignments when they occur in ini files.

The macro arguments are as follows:

- *ID* is a C++ identifier that becomes the name of a global variable, a pointer to a `cConfigOption` object that the macro creates. It lets you refer to the configuration option, e.g. when querying its value using `cConfiguration`'s member functions.
- *NAME* is the name of the option (a string).
- *KIND* applies to per-object configuration options, and clarifies what kind of objects the option applies to. Its value must be one of: `KIND_COMPONENT` (module or channel), `KIND_CHANNEL`, `KIND_MODULE` (simple or compound module), `KIND_SIMPLE_MODULE`, `KIND_PARAMETER` (module or channel parameter), `KIND_STATISTIC` (statistic declared in NED via `@statistic`), `KIND_SCALAR` (output scalar), `KIND_VECTOR` (output vector), `KIND_UNSPECIFIED_TYPE` (only used for the **`typename`** option), `KIND_OTHER` (anything else).
- *TYPE* is the data type of the config option; it must be one of: `CFG_BOOL`, `CFG_INT`, `CFG_DOUBLE`, `CFG_STRING`, `CFG_FILENAME`, `CFG_FILENAMES`, `CFG_PATH`, `CFG_CUSTOM`. The most significant difference between filesystem-related types (filename, filenames, path) and plain strings is that relative filenames and paths are automatically converted to absolute when the configuration is read, with the base directory being the location of the ini file where the configuration entry was read from.
- *UNIT* is a string that names the measurement unit in which the option's value is to be interpreted; it implies type `CFG_DOUBLE`.
- *DEFAULTVALUE* is the default value in textual form (string); this should be `nullptr` if the option has no default value.
- *DESCRIPTION* is an arbitrarily long string that describes the purpose and and operation of the option. It will be used in help texts etc.

For example, the `debug-on-errors` option is declared in the following way:

```
Register_GlobalConfigOption(CFGID_DEBUG_ON_ERRORS, "debug-on-errors",
    CFG_BOOL, "false", "When enabled, runtime errors will cause...");
```

The macro will register the option, and also declare the `CFGID_DEBUG_ON_ERRORS` variable as pointer to a `cConfigOption`. The variable can be used later as a "handle" when reading the option value from the configuration database.

17.2.2 Reading the Value

The configuration is accessible via the `getConfig()` method of `cEnvir`. It returns a pointer to the configuration object (`cConfiguration`):

```
cConfiguration *config = getEnvir()->getConfig();
```

`cConfiguration` provides several methods for querying the configuration.

> **NOTE:** The configuration object provides a flattened view of the ini file. Sections inheriting from each other are merged. Configuration options provided on the command line in the form `-option=value` are added first to the object. This ensures that the command line options take precedence over the values specified in the INI file.

```
const char *getAsCustom(cConfigOption *entry, const char *fallbackValue=nullptr);
bool getAsBool(cConfigOption *entry, bool fallbackValue=false);
long getAsInt(cConfigOption *entry, long fallbackValue=0);
double getAsDouble(cConfigOption *entry, double fallbackValue=0);
std::string getAsString(cConfigOption *entry, const char *fallbackValue="");
std::string getAsFilename(cConfigOption *entry);
std::vector<std::string> getAsFilenames(cConfigOption *entry);
std::string getAsPath(cConfigOption *entry);
```

fallbackValue is returned if the value is not specified in the configuration, and there is no default value.

```
bool debug = getEnvir()->getConfig()->getAsBool(CFGID_PARSIM_DEBUG);
```

17.3 Simulation Lifetime Listeners

`cISimulationLifecycleListener` is a callback interface for receiving notifications at various stages of simulations: setting up, running, tearing down, etc. Extension classes such as custom event schedulers often need this functionality for performing initalization and various other tasks.

Listeners of the type `cISimulationLifecycleListener` need to be added to `cEnvir` with its `addLifecycleListener()` method, and removed with `removeLifecycleListener()`.

```
cISimulationLifecycleListener *listener = ...;
getEnvir()->addLifecycleListener(listener);
// and finally:
getEnvir()->removeLifecycleListener(listener);
```

To implement a simulation lifecycle listener, subclass from `cISimulationLifecycleListener`, and override its `lifecycleEvent()` method. It has the following signature:

```
virtual void lifecycleEvent(SimulationLifecycleEventType eventType, cObject *detai
```

Event type is one of the following. Their names are fairly self-describing, but the API documentation contains more precise information.

- `LF_ON_STARTUP`
- `LF_PRE_NETWORK_SETUP`, `LF_POST_NETWORK_SETUP`
- `LF_PRE_NETWORK_INITIALIZE`, `LF_POST_NETWORK_INITIALIZE`
- `LF_ON_SIMULATION_START`
- `LF_ON_SIMULATION_PAUSE`, `LF_ON_SIMULATION_RESUME`
- `LF_ON_SIMULATION_SUCCESS`, `LF_ON_SIMULATION_ERROR`
- `LF_PRE_NETWORK_FINISH`, `LF_POST_NETWORK_FINISH`

- LF_ON_RUN_END
- LF_PRE_NETWORK_DELETE, LF_POST_NETWORK_DELETE
- LF_ON_SHUTDOWN

The *details* argument is currently nullptr; further OMNEST versions may pass extra information in it. Notifications always refer to the active simulation in case there're more (see cSimulation's getActiveSimulation()).

Simulation lifecycle listeners are mainly intended for use by classes that extend the simulator's functionality, for example custom event schedulers and output vector/scalar managers. The lifecycle of such an extension object is managed by OMNEST, so one can use their constructor to create and add the listener object to cEnvir, and the destructor to remove and delete it. The code is further simplified if the extension object itself implements cISimulationLifecycleListener:

```
class CustomScheduler : public cScheduler, public cISimulationLifecycleListener
{
  public:
    CustomScheduler() { getEnvir()->addLifecycleListener(this); }
    ~CustomScheduler() { getEnvir()->removeLifecycleListener(this); }
    //...
};
```

17.4 cEvent

cEvent represents an event in the discrete event simulator. When events are scheduled, they are inserted into the future events set (FES). During the simulation, events are removed from the FES and executed one by one in timestamp order. A cEvent is executed by invoking its execute() member function. execute() should be overridden in subclasses to carry out the actions associated with the event.

> **NOTE:** cMessage is also a subclass of cEvent. Its execute() method calls the handleMessage() method of the message's destination module, or switches to the coroutine of its activity() method.

execute() has the following signature:

```
virtual void execute() = 0;
```

Raw (non-message) event objects are an internal mechanism of the OMNEST simulation kernel, and should not used in programming simulation models. However, they can be very useful when implementing custom event schedulers. For example, in co-simulation, events that occur in the other simulator may be represented with a cEvent in OMNEST. Simulation time limit is also implemented with a custom cEvent.

17.5 Defining a New Random Number Generator

This interface lets one add new RNG implementations (see section 7.3) to OMNEST. The motivation might be achieving integration with external software (for example something like

Akaroa), or exactly replicating the trajectory of a simulation ported from another simulation framework that uses a different RNG.

The new RNG C++ class must implement the `cRNG` interface, and can be activated with the **`rng-class`** configuration option.

17.6 Defining a New Event Scheduler

This extension interface lets one replace the event scheduler class with a custom one, which is the key for implementing many features including cosimulation, real-time simulation, network or device emulation, and distributed simulation.

The job of the event scheduler is to always return the next event to be processed by the simulator. The default implementation returns the first event in the future events list. Other variants:

- For real-time simulation, this scheduler is replaced with one augmented with *wait* calls (e.g. `usleep()`) that synchronize the simulation time to the system clock. There are several options on what should happen if the simulation time has already fallen behind: one may re-adjust the reference time, leave it unchanged in the hope of catching up later, or stop with an error message.
- For emulation, the real-time scheduler is augmented with code that captures packets from real network devices, and inserts them into the simulation. INET Framework, the main protocol simulation package for OMNEST, contains an emulation scheduler. It uses the *pcap* library to capture packets, and raw sockets to send packets to a real network device. Emulation in INET also involves *header serializer* classes that convert between protocol headers and their C++ object representations used within the simulation.
- For parallel simulation (see chapter 16), the scheduler is modified to listen for messages arriving from other logical processes (LPs), and inserts them into the simulation. The scheduler also blocks the simulation when it is not safe to execute the next event due to potential causality violation, until clearance arrives from other LPs to continue in the form of a null message.
- OMNEST supports distributed simulation using HLA (IEEE 1516) [1] as well. The scheduler plays the role of the HLA Federate Ambassador, is responsible for exchanging messages (interactions, change notifications, etc.) with other federates, and performs time regulation.
- OMNEST also supports mixing SystemC (IEEE 1666-2005) modules with OMNEST modules in the simulation. When this feature is enabled, there are two future event lists in the simulation, OMNEST's and SystemC's, and a special scheduler takes care that events are consumed from both lists in increasing timestamp order. This method of performing mixed simulations is orders of magnitude faster and also more flexible than letting the two simulators execute in separate processes and communicate over a pipe or socket connection.

The scheduler C++ class must implement the `cScheduler` interface, and can be activated with the **`scheduler-class`** configuration option.

[1] The source code for the HLA and SystemC integration features are not open source, but they are available to researchers on request free of charge.

Simulation lifetime listeners and the `cEvent` class can be extremely useful when implementing certain types of event schedulers.

To see examples of scheduler classes, check the `cSequentialScheduler` and `cRealTimeScheduler` classes in the simulation kernel, `cSocketRTScheduler` which is part of the *Sockets* sample simulation, or `cParsimSynchronizer` and its subclasses that are part of the parallel simulation support of OMNEST.

17.7 Defining a New FES Data Structure

This extension interface allows one to replace the data structure used for storing future events during simulation, i.e. the FES. Replacing the FES may make sense for specialized workloads, or for the purpose of performance comparison of various FES algorithms. (The default, binary heap based FES implementation is a good choice for general workloads.)

The FES C++ class must implement the `cFutureEventSet` interface, and can be activated with the **`futureeventset-class`** configuration option.

17.8 Defining a New Fingerprint Algorithm

This extension interface allows one to replace or extend the fingerprint computation algorithm (see section 15.4).

The fingerprint computation class must implement the `cFingerprintCalculator` interface, and can be activated with the **`fingerprintcalculator-class`** configuration option.

17.9 Defining a New Output Scalar Manager

An output scalar manager handles the recording the scalar and histogram output data. The default output scalar manager is `cFileOutputScalarManager` that saves data into `.sca` files. This extension interface allows one to create additional means of saving scalar and histogram results, for example database or CSV output.

The new class must implement `cIOutputScalarManager`, and can be activated with the **`outputscalarmanager-class`** configuration option.

17.10 Defining a New Output Vector Manager

An output vector manager handles the recording output vectors, produced for example by `cOutVector` objects. The default output vector manager is `cIndexedFileOutputVectorManager` that saves data into `.vec` files, indexed in separate `.vci` files. This extension interface allows one to create additional means of saving vector results, for example database or CSV output.

The new class must implement the `cIOutputVectorManager` interface, and can be activated with the **`outputvectormanager-class`** configuration option.

17.11 Defining a New Eventlog Manager

An eventlog manager handles the recording of simulation history into an event log (see 13). The default eventlog manager is `EventlogFileManager`, which records into file, and also allows for some filtering. By replacing the default eventlog manager class, one can introduce additional filtering, record into a different file format or to different storage (e.g. to a database or a remote vizualizer).

The new class must implement the `cIEventlogManager` interface, and can be activated with the **`eventlogmanager-class`** configuration option.

17.12 Defining a New Snapshot Manager

A snapshot manager provides an output stream to which snapshots are written (see section 7.11.5). The default snapshot manager is `cFileSnapshotManager`.

The new class must implement the `cISnapshotManager` interface, and can be activated with the **`snapshotmanager-class`** configuration option.

17.13 Defining a New Configuration Provider

17.13.1 Overview

The configuration provider extension lets one replace ini files with some other storage implementation, for example a database. The configuration provider C++ class must implement the `cConfigurationEx` interface, and can be activated with the **`configuration-class`** configuration option.

The `cConfigurationEx` interface abstracts the inifile-based data model to some degree. It assumes that the configuration data consists of several *named configurations*. Before every simulation run, one of the *named configurations* is activated, and from then on, all queries into the configuration operate on the *active named configuration* only.

It practice, you will probably use the `SectionBasedConfiguration` class (in `src/envir`) or subclass from it, because it already implements a lot of functionality that you would otherwise have to.

`SectionBasedConfiguration` does not assume ini files or any other particular storage format; instead, it accepts an object that implements the `cConfigurationReader` interface to provide the data in raw form to it. The default implementation of `cConfigurationReader` is `InifileReader`.

17.13.2 The Startup Sequence

From the configuration extension's point of view, the startup sequence looks like the following (see `src/envir/startup.cc` in the source code):

1. First, ini files specified on the command-line are read into a *boot-time configuration object*. The boot-time configuration is always a `SectionBasedConfiguration` with `InifileReader`.

2. Shared libraries are loaded (see the `-l` command-line option, and the **`load-libs`** configuration option). This allows configuration classes to come from shared libraries.
3. The **`configuration-class`** configuration option is examined. If it is present, a configuration object of the given class is instantiated, and replaces the boot-time configuration. The new configuration object is initialized from the boot-time configuration, so that it can read parameters (e.g. database connection parameters, XML file name, etc) from it. Then the boot-time configuration object is deallocated.
4. The **`load-libs`** option from the new configuration object is processed.
5. Then everything goes on as normally, using the new configuration object.

17.13.3 Providing a Custom Configuration Class

To replace the configuration object with a custom implementation, one needs to subclass `cConfigurationEx`, register the new class,

```
#include "cconfiguration.h"

class CustomConfiguration : public cConfigurationEx
{
   ...
};

Register_Class(CustomConfiguration);
```

and then activate it in the boot-time configuration:

```
[General]
configuration-class = CustomConfiguration
```

17.13.4 Providing a Custom Reader for SectionBasedConfiguration

As said already, writing a configuration class from scratch can be a lot of work, and it may be more practical to reuse `SectionBasedConfiguration` with a different configuration reader class. This can be done with **`sectionbasedconfig-configreader-class`** config option, which is interpreted by `SectionBasedConfiguration`. Specify the following in the boot-time ini file:

```
[General]
configuration-class = SectionBasedConfiguration
sectionbasedconfig-configreader-class = <new-reader-class>
```

The configuration reader class should look like this:

```
#include "cconfigreader.h"

class DatabaseConfigurationReader : public cConfigurationReader
{
   ...
};

Register_Class(DatabaseConfigurationReader);
```

17.14 Implementing a New User Interface

It is possible to extend OMNEST with a new user interface. The new user interface will have fully equal rights to Cmdenv and Qtenv; that is, it can be activated by starting the simulation executable with the `-u` *<name>* command-line or the **`user-interface`** configuration option, it can be made the default user interface, it can define new command-line options and configuration options, and so on.

User interfaces must implement (i.e. subclass from) `cRunnableEnvir`, and must be registered to OMNEST with the `Register_OmnetApp()` macro. In practice, you will almost always want to subclass `EnvirBase` instead of `cRunnableEnvir`, because `EnvirBase` already implements lots of functionality that otherwise you'd have to.

> **NOTE:** If you want something completely different from what `EnvirBase` provides, such as embedding the simulation kernel into another application, then you should be reading section 18.2, not this one.

An example user interface:

```
#include "envirbase.h"

class FooEnv : public EnvirBase
{
    ...
};

Register_OmnetApp("FooEnv", FooEnv, 30, "an experimental user interface");
```

The `envirbase.h` header comes from the `src/envir` directory, so it is necessary to add it to the include path (`-I`).

The arguments to `Register_OmnetApp()` include the user interface name (for use with the `-u` and **`user-interface`** options), the C++ class that implements it, a weight for default user interface selection (if `-u` is missing, the user interface with the largest weight will be activated), and a description string (for help and other purposes).

The C++ class should implement all methods left pure virtual in `EnvirBase`, and possibly others if you want to customize their behavior. One method that you will surely want to reimplement is `run()` – this is where your user interface runs. When this method exits, the simulation program exits.

> **NOTE:** A good starting point for implementing your own user interface is Cmdenv – just copy and modify its source code to quickly get going.

Chapter 18

Embedding the Simulation Kernel

18.1 Architecture

OMNEST has a modular architecture. The following diagram illustrates the high-level architecture of OMNEST simulations:

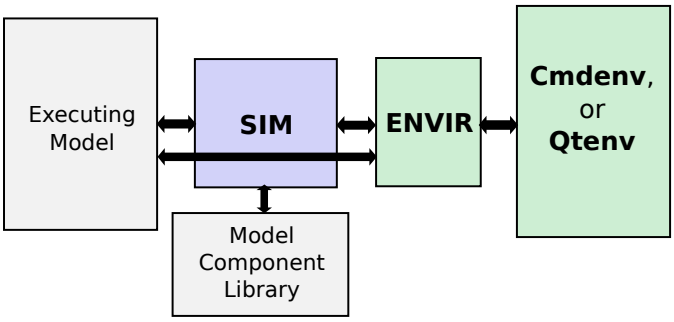


Figure 18.1: The architecture of OMNEST simulations

The blocks represent the following components:

- **Sim** is the simulation kernel and class library. Sim is a library linked to simulation programs.
- **Envir** is another library that contains all code that is common to all the user interfaces. `main()` also resides in the Envir library. Envir presents itself towards Sim and the executing model as an instance of the `cEnvir` facade class. Some aspects of the Envir library like result recording can be customized using plugin interfaces. Embedding OMNEST into applications usually involves writing a custom `cEnvir` subclass (see sections 17.14 and 18.2.)
- **Cmdenv, Qtenv** are Envir-based libraries that contain specific user interface implementations. A simulation program is linked with one or more of them; in the latter case, one of the UI libraries is chosen and instantiated either explicitly or automatically when the program starts.

- The **Model Component Library** includes simple module definitions and their C++ implementations, compound module types, channels, networks, message types, and everything belonging to models that have been linked to the simulation program. A simulation program can run any model that contains all of the required linked components.
- The **Executing Model** is the model that is set up for simulation. This model contains objects (modules, channels, and so on) that are all instances of the components in the model component library.

The arrows in the figure describe how components interact with each other:

- **Executing Model** ⇔ **Sim**. The simulation kernel manages the future events and activates modules in the executing model as events occur. The modules of the executing model are stored in an instance of the class `cSimulation`. In turn, the executing model calls functions in the simulation kernel and uses classes in the Sim library.
- **Sim** ⇔ **Model Component Library**. The simulation kernel instantiates simple modules and other components when the simulation model is set up at the beginning of the simulation run. In addition, it refers to the component library when dynamic module creation is used. The mechanisms for registering and looking up components in the model component library are implemented as part of Sim.
- **Executing Model** ⇔ **Envir**. The Envir presents itself as a facade object towards the executing model. Model code directly accesses Envir e.g. for logging (`EV«`).
- **Sim** ⇔ **Envir**. Envir is in full command of what happens in the simulation program. Envir contains the `main()` function where execution begins. Envir determines which models should be set up for simulation, and instructs Sim to do so. Envir contains the main simulation loop (*determine-next-event*, *execute-event* sequence) and invokes the simulation kernel for the necessary functionality (event scheduling and event execution are implemented in Sim). Envir catches and handles errors and exceptions that occur in the simulation kernel or in the library classes during execution. Envir presents a single facade object toward Sim – no Envir internals are visible to Sim or the executing model. During simulation model setup, Envir supplies module parameter values for Sim when Sim asks for them. Sim writes output vectors via Envir, so one can redefine the output vector storing mechanism by changing Envir. Sim and its classes use Envir to print debug information.
- **Envir** ⇔ **Cmdenv/Qtenv**. Cmdenv, and Qtenv are concrete user interface implementations. When a simulation program is started, the `main()` function (which is part of Envir) determines the appropriate user interface class, creates an instance and runs it. Sim's or the model's calls on Envir are delegated to the user interface.

18.2 Embedding the OMNEST Simulation Kernel

This section discusses the issues of embedding the simulation kernel or a simulation model into a larger application. We assume that you do not just want to change one or two aspects of the simulator (such as , event scheduling or result recording) or create a new user interface similar to Cmdenv or Qtenv – if so, see chapter 17.

For the following section, we assume that you will write the embedding program from scratch, that is, starting from a `main()` function.

18.2.1 The main() Function

The minimalistic program described below initializes the simulation library and runs two simulations. In later sections we will review the details of the code and discuss how to improve it.

```
#include <omnetpp.h>
using namespace omnetpp;

int main(int argc, char *argv[])
{
    // the following line MUST be at the top of main()
    cStaticFlag dummy;

    // initializations
    CodeFragments::executeAll(CodeFragments::STARTUP);
    SimTime::setScaleExp(-12);

    // load NED files
    cSimulation::loadNedSourceFolder("./foodir");
    cSimulation::loadNedSourceFolder("./bardir");
    cSimulation::doneLoadingNedFiles();

    // run two simulations
    simulate("FooNetwork", 1000);
    simulate("BarNetwork", 2000);

    // deallocate registration lists, loaded NED files, etc.
    CodeFragment::executeAll(CodeFragment::SHUTDOWN);
    return 0;
}
```

The first few lines of the code initialize the simulation library. The purpose of `cStaticFlag` is to set a global variable to `true` for the duration of the `main()` function, to help the simulation library handle exceptions correctly in extreme cases. `CodeFragment::executeAll(CodeFragment::STA` performs various startup tasks, such as building registration tables out of the `Define_Module()`, `Register_Class()` and similar entries throughout the code. `SimTime::setScaleExp(-12)` sets the simulation time resolution to picoseconds; other values can be used as well, but it is mandatory to choose one.

> **NOTE:** The simulation time exponent cannot be changed at a later stage, since it is a global variable, and the values of the existing `simtime_t` instances would change.

The code then loads the NED files from the `foodir` and `bardir` subdirectories of the working directory (as if the NED path was `./foodir;./bardir`), and runs two simulations.

18.2.2 The simulate() Function

A minimalistic version of the `simulate()` function is shown below. In order to shorten the code, the exception handling code has been ommited (`try/catch` blocks) apart from the event loop. However, every line is marked with "`E!`" where various problems with the simulation model can occur and can be thrown as exceptions.

```
void simulate(const char *networkName, simtime_t limit)
{
    // look up network type
    cModuleType *networkType = cModuleType::find(networkName);
    if (networkType == nullptr) {
        printf("No such network: %s\n", networkName);
        return;
    }

    // create a simulation manager and an environment for the simulation
    cEnvir *env = new CustomSimulationEnv(argc, argv, new EmptyConfig());
    cSimulation *sim = new cSimulation("simulation", env);
    cSimulation::setActiveSimulation(sim);

    // set up network and prepare for running it
    sim->setupNetwork(networkType); //E!
    sim->setSimulationTimeLimit(limit);

    // prepare for running it
    sim->callInitialize();

    // run the simulation
    bool ok = true;
    try {
        while (true) {
            cEvent *event = sim->takeNextEvent();
            if (!event)
                break;
            sim->executeEvent(event);
        }
    }
    catch (cTerminationException& e) {
        printf("Finished: %s\n", e.what());
    }
    catch (std::exception& e) {
        ok = false;
        printf("ERROR: %s\n", e.what());
    }

    if (ok)
        sim->callFinish();  //E!

    sim->deleteNetwork();  //E!

    cSimulation::setActiveSimulation(nullptr);
    delete sim; // deletes env as well
}
```

The function accepts a network type name (which must be fully qualified with a package name) and a simulation time limit.

In the first few lines, the code looks up the network among the available module types, and prints an error message if it is not found.

Then it proceeds to create and activate a simulation manager object (`cSimulation`). The simulation manager requires another object, called the environment object. The environment object is used by the simulation manager to read the configuration. In addition, the simulation results are also written via the environment object.

The environment object (`CustomSimulationEnv` in the above code) must be provided by the programmer; this is described in detail in a later section.

> **NOTE:** In versions 4.x and earlier, the simulation manager and the environment object could be accessed as `simulation` and `ev` (which were global variables in 3.x and macros in 4.x). In 5.x they can be accessed with the `getSimulation()` and `getEnvir()` functions, which are basically aliases to `cSimulation::getActiveSimulation()` and `cSimulation::getActiveSimulation()->getEnvir()`.

The network is then set up in the simulation manager. The `sim->setupNetwork()` method creates the system module and recursively all modules and their interconnections; module parameters are also read from the configuration (where required) and assigned. If there is an error (for example, module type not found), an exception will be thrown. The exception object is some kind of `std::exception`, usually a `cRuntimeError`.

If the network setup is successful, `sim->callInitialize()` is invoked next, to run the initialization code of modules and channels in the network. An exception is thrown if something goes wrong in any of the `initialize()` methods.

The next lines run the simulation by calling `sim->takeNextEvent()` and `sim->executeEvent()` in a loop. The loop is exited when an exception occurs. The exception may indicate a runtime error, or a normal termination condition such as when there are no more events, or the simulation time limit has been reached. (The latter are represented by `cTerminationException`.)

If the simulation has completed successfully (`ok==true`), the code goes on to call the `finish()` methods of modules and channels. Then, regardless whether there was an error, cleanup takes place by calling `sim->deleteNetwork()`.

Finally, the simulation manager object is deallocated, but the active simulation manager is not allowed to be deleted; therefore it is deactivated using `setActiveSimulation(nullptr)`.

18.2.3 Providing an Environment Object

The environment object needs to be subclassed from the `cEnvir` class, but since it has many pure virtual methods, it is easier to begin by subclassing `cNullEnvir`. `cNullEnvir` defines all pure virtual methods with either an empty body or with a body that throws an `"unsupported method called"` exception. You can redefine methods to be more sophisticated later on, as you progress with the development.

You must redefine the `readParameter()` method. This enables module parameters to obtain their values. For debugging purposes, you can also redefine `sputn()` where module log messages are written to. `cNullEnvir` only provides one random number generator, so if your simulation model uses more than one, you also need to redefine the `getNumRNGs()` and `getRNG(k)` methods. To print or store simulation records, redefine `recordScalar()`, `recordStatistic()` and/or the output vector related methods. Other `cEnvir` methods are invoked from the simulation kernel to inform the environment about messages being sent, events scheduled and cancelled, modules created, and so on.

The following example shows a minimalistic environment class that is enough to get started:

```
class CustomSimulationEnv : public cNullEnvir
```

```
{
  public:
    // constructor
    CustomSimulationEnv(int ac, char **av, cConfiguration *c) :
        cNullEnvir(ac, av, c) {}

    // model parameters: accept defaults
    virtual void readParameter(cPar *par) {
        if (par->containsValue())
            par->acceptDefault();
        else
            throw cRuntimeError("no value for %s", par->getFullPath().c_str());
    }

    // send module log messages to stdout
    virtual void sputn(const char *s, int n) {
        (void) ::fwrite(s,1,n,stdout);
    }
};
```

18.2.4 Providing a Configuration Object

The configuration object needs to subclass from `cConfiguration`. `cConfiguration` also has several methods, but the typed ones (`getAsBool()`, `getAsInt()`, etc.) have default implementations that delegate to the much fewer string-based methods (`getConfigValue()`, etc.).

It is fairly straightforward to implement a configuration class that emulates an empty ini file:

```
class EmptyConfig : public cConfiguration
{
  protected:
    class NullKeyValue : public KeyValue {
      public:
        virtual const char *getKey() const {return nullptr;}
        virtual const char *getValue() const {return nullptr;}
        virtual const char *getBaseDirectory() const {return nullptr;}
    };
    NullKeyValue nullKeyValue;

  protected:
    virtual const char *substituteVariables(const char *value) {return value;}

  public:
    virtual const char *getConfigValue(const char *key) const
        {return nullptr;}
    virtual const KeyValue& getConfigEntry(const char *key) const
        {return nullKeyValue;}
    virtual const char *getPerObjectConfigValue(const char *objectFullPath,
        const char *keySuffix) const {return nullptr;}
    virtual const KeyValue& getPerObjectConfigEntry(const char *objectFullPath,
        const char *keySuffix) const {return nullKeyValue;}
};
```

18.2.5 Loading NED Files

NED files can be loaded with any of the following static methods of `cSimulation`: `loadNedSourceFolder()`, `loadNedFile()`, and `loadNedText()`. The first method loads an entire subdirectory tree, the second method loads a single NED file, and the third method takes a literal string containing NED code and parses it.

> **NOTE:** One use of `loadNedText()` is to parse NED sources previously converted to C++ string constants and linked into the executable. This enables creating executables that are self-contained, and do not require NED files to be distributed with them.

The above functions can also be mixed, but after the last call, `doneLoadingNedFiles()` must be invoked (it checks for unresolved NED types).

Loading NED files has a global effect; therefore they cannot be unloaded.

18.2.6 How to Eliminate NED Files

It is possible to get rid of NED files altogether. This would also remove the dependency on the `oppnedxml` library and the code in `sim/netbuilder`, although at the cost of additional coding.

> **NOTE:** When the only purpose is to get rid of NED files as external dependency of the program, it is simpler to use `loadNedText()` on NED files converted to C++ string constants instead.

The trick is to write `cModuleType` and `cChannelType` objects for simple module, compound module and channel types, and register them manually. For example, `cModuleType` has pure virtual methods called `createModuleObject()`, `addParametersAndGatesTo(module)`, `setupGateVectors(module)`, `buildInside(module)`, which you need to implement. The body of the `buildInside()` method would be similar to C++ files generated by `nedtool` of OMNEST 3.x.

18.2.7 Assigning Module Parameters

As already mentioned, modules obtain values for their input parameters by calling the `readParameter()` method of the environment object (`cEnvir`).

> **NOTE:** `readParameter()` is only called for parameters that have not been set to a fixed (i.e. non-`default`) value in the NED files.

The `readParameter()` method should be written in a manner that enables it to assign the parameter. When doing so, it can recognize the parameter from its name (`par->getName()`), from its full path (`par->getFullPath()`), from the owner module's class (`par->getOwner()->getClassName()`) or NED type name (`((cComponent *)par->getOwner())->getNedTypeName()`). Then it can set the parameter using one of the typed setter methods (`setBoolValue()`, `setLongValue()`, etc.), or set it to an expression provided in string form (`parse()` method). It can also accept the default value if it exists (`acceptDefault()`).

The following code is a straightforward example that answers parameter value requests from a pre-filled table.

```
class CustomSimulationEnv : public cNullEnvir
{
  protected:
    // parameter (fullpath,value) pairs, needs to be pre-filled
    std::map<std::string,std::string> paramValues;
  public:
    ...
    virtual void readParameter(cPar *par) {
        if (paramValues.find(par->getFullPath())!=paramValues.end())
            par->parse(paramValues[par->getFullPath()]);
        else if (par->containsValue())
            par->acceptDefault();
        else
            throw cRuntimeError("no value for %s", par->getFullPath().c_str());
    }
};
```

18.2.8 Extracting Statistics from the Model

There are several ways you can extract statistics from the simulation.

C++ Calls into the Model

Modules in the simulation are C++ objects. If you add the appropriate public getter methods to the module classes, you can call them from the main program to obtain statistics. Modules may be looked up with the `getModuleByPath()` method of `cSimulation`, then cast to the specific module type via `check_and_cast<>()` so that the getter methods can be invoked.

```
cModule *mod = getSimulation()->getModuleByPath("Network.client[2].app");
WebApp *appMod = check_and_cast<WebApp *>(mod);
int numRequestsSent = appMod->getNumRequestsSent();
double avgReplyTime = appMod->getAvgReplyTime();
...
```

The drawback of this approach is that getters need to be added manually to all affected module classes, which might not be practical, especially if modules come from external projects.

`cEnvir` Callbacks

A more general way is to catch `recordScalar()` method calls in the simulation model. The `cModule`'s `recordScalar()` method delegates to the similar function in `cEnvir`. You may define the latter function so that it stores all recorded scalars (for example in an `std::map`), where the main program can find them later. Values from output vectors can be captured in a similar manner.

An example implementation:

```
class CustomSimulationEnv : public cNullEnvir
{
  private:
    std::map<std::string, double> results;
```

```
  public:
    virtual void recordScalar(cComponent *component, const char *name,
                              double value, opp_string_map *attributes=nullptr)
    {
       results[component->getFullPath()+"."+name] = value;
    }

    const std::map<std::string, double>& getResults() {return results;}
};

...

const std::map<std::string, double>& results = env->getResults();
int numRequestsSent = results["Network.client[2].app.numRequestsSent"];
double avgReplyTime = results["Network.client[2].app.avgReplyTime"];
```

A drawback of this approach is that compile-time checking of statistics names is lost, but the advantages are that any simulation model can now be used without changes, and that capturing additional statistics does not require code modification in the main program.

18.2.9 The Simulation Loop

To run the simulation, the `takeNextEvent()` and `executeEvent()` methods of `cSimulation` must be called in a loop:

```
cSimulation *sim = getSimulation();
while (sim->getSimTime() < limit) {
    cEvent *event = sim->takeNextEvent();
    sim->executeEvent(event);
}
```

Depending on the concrete scheduler class, the `takeNextEvent()` may return `nullptr` in certain cases. The default `cSequentialScheduler` never returns `nullptr`.

The execution may terminate in various ways. Runtime errors cause a `cRuntimeError` (or other kind of `std::exception`) to be thrown. `cTerminationException` is thrown on normal termination conditions, such as when the simulation runs out of events to process.

You may customize the loop to exit on other termination conditions as well, such as on a simulation time limit (see above), on a CPU time limit, or when results reach a required accuracy. It is relatively straightforward to build in progress reporting and interactivity (start/stop).

Animation can be hooked up to the appropriate callback methods of `cEnvir`: `beginSend()`, `sendHop()`, `endSend()`, and others.

18.2.10 Multiple, Coexisting Simulations

It is possible for several instances of `cSimulation` to coexist, and also to set up and simulate a network in each instance. However, this requires frequent use of `cSimulation::setActiveSimulation()`. Before invoking any `cSimulation` method or module method, the corresponding `cSimulation` instance needs to be designated as the active simulation manager.

Every `cSimulation` instance should have its own associated environment object (`cEnvir`). Environment objects may not be shared among several `cSimulation` instances. The `cSimulation`'s destructor also removes the associated `cEnvir` instance.

`cSimulation` instances may be reused from one simulation to another, but it is also possible to create a new instance for each simulation run.

> **NOTE:** It is not possible to run different simulations concurrently from different theads, due to the use of global variables which are not easy to eliminate, such as the active simulation manager pointer and the active environment object pointer. Static buffers and objects (like string pools) are also used for efficiency reasons in some places inside the simulation kernel.

18.2.11 Installing a Custom Scheduler

The default event scheduler is `cSequentialScheduler`. To replace it with a different scheduler (e.g. `cRealTimeScheduler` or your own scheduler class), add a `setScheduler()` call into `main()`:

```
cScheduler *scheduler = new CustomScheduler();
getSimulation()->setScheduler(scheduler);
```

It is usually not a good idea to change schedulers in the middle of a simulation, therefore `setScheduler()` may only be called when no network is set up.

18.2.12 Multi-Threaded Programs

The OMNEST simulation kernel is not reentrant; therefore it must be protected against concurrent access.

Appendix A

NED Reference

A.1 Syntax

A.1.1 NED File Name Extension

NED files have the `.ned` file name suffix. This is mandatory, and cannot be overridden.

A.1.2 NED File Encoding

NED files are ASCII, but non-ASCII characters are permitted in comments and string literals. This allows for using encodings that are a superset of ASCII, for example ISO 8859-1 and UTF-8.

> **NOTE:** There is no standard way to specify or determine the encoding of a NED file. It is up to the user to configure the desired encoding in text editors and other tools that edit or process NED files.

String literals (e.g. in parameter values) will be passed to the C++ code as `const char *` without any conversion; it is up to the simulation model to interpret them using the desired encoding.

Line ending may be either CR or CRLF, regardless of the platform.

A.1.3 Reserved Words

The following words are reserved, and cannot be used for identifiers:

```
allowunconnected bool channel channelinterface connections const default dou-
ble extends false for gates if import index inf inout input int like module mod-
uleinterface nan network null nullptr object output package parameters parent
property simple sizeof string submodules this true typename types undefined volatile
xml xmldoc
```

A.1.4 Identifiers

Identifiers must be composed of letters of the English alphabet (a-z, A-Z), numbers (0-9) and underscore "_". Identifiers may only begin with a letter or underscore.

The recommended way to compose identifiers from multiple words is to capitalize the beginning of each word (*camel case*).

A.1.5 Case Sensitivity

Keywords and identifiers in the NED language are case sensitive. For example, `TCP` and `Tcp` are two different names.

A.1.6 Literals

String Literals

String literals use double quotes. The following C-style backslash escapes are recognized: `\b`, `\f`, `\n`, `\r`, `\t`, `\\`, `\"`, and `\xhh` where *h* is a hexadecimal digit.

Numeric Constants

Numeric constants are accepted in the usual decimal, hexadecimal (`0x` prefix) and scientific notations. Octal numbers are not accepted (numbers that start with the `0` digit are interpreted as decimal.)

`nan`, **`inf`** and –**`inf`** mean the floating-point not-a-number, positive infinity and negative infinity values, respectively.

Quantity Constants

A quantity constant has the form *(<numeric-constant> <unit>)+*, for example `12.5mW` or `3h 15min 37.2s`. Whitespace is optional in front of a unit, but must be present after a unit if it is followed by a number.

When multiple measurement units are present, they have to be convertible into each other (i.e. refer to the same physical quantity).

Section A.5.11 lists the units recognized by OMNEST. Other units can be used as well; the only downside being that OMNEST will not be able to perform conversions on them.

Null Object References

The keywords **`null`** and **`nullptr`** are synonymous, and denote an object reference that doesn't refer to any valid object.

Undefined

The keyword **`undefined`** denotes the "missing value" value, similar to `(void)0` in C/C++. **`undefined`** has its own type, and cannot be cast to any other type.

A.1.7 Comments

Comments can be placed at the end of lines. Comments begin with a double slash `//`, and continue until the end of the line.

A.1.8 Grammar

The grammar of the NED language can be found in Appendix B.

A.2 Built-in Definitions

The NED language has the following built-in definitions, all in the `ned` package: channels `IdealChannel`, `DelayChannel`, and `DatarateChannel`; module interfaces `IBidirectionalChannel`, and `IUnidirectionalChannel`. The latter two are reserved for future use.

The bodies of `@statistic` properties have been omitted for brevity from the following listing.

> **NOTE:** One can print the full definitions by running `opp_run -h neddecls`.

```
package ned;
@namespace("omnetpp");

channel IdealChannel
{
    @class(cIdealChannel);
}

channel DelayChannel
{
    @class(cDelayChannel);
    @signal[messageSent](type=omnetpp::cMessage);
    @signal[messageDiscarded](type=omnetpp::cMessage);
    @statistic[messages](source="constant1(messageSent)";record=count?;interpolati
    @statistic[messagesDiscarded](source="constant1(messageDiscarded)";record=coun
    bool disabled @mutable = default(false);
    double delay @mutable = default(0s) @unit(s); // propagation delay
}

channel DatarateChannel
{
    @class(cDatarateChannel);
    @signal[channelBusy](type=long);
    @signal[messageSent](type=omnetpp::cMessage);
    @signal[messageDiscarded](type=omnetpp::cMessage);
    @statistic[busy](source=channelBusy;record=vector?;interpolationmode=sample-ho
    @statistic[utilization](source="timeavg(channelBusy)";record=last?);
    @statistic[packets](source="constant1(messageSent)";record=count?;interpolatio
    @statistic[packetBytes](source="packetBytes(messageSent)";record=sum?;unit=B;i
    @statistic[packetsDiscarded](source="constant1(messageDiscarded)";record=count
```

```
    @statistic[throughput](source="sumPerDuration(packetBits(messageSent))";record
    bool disabled @mutable = default(false);
    double delay @mutable = default(0s) @unit(s); // propagation delay
    double datarate @mutable = default(0bps) @unit(bps); // bits per second; 0=inf
    double ber @mutable = default(0); // bit error rate (BER)
    double per @mutable = default(0); // packet error rate (PER)
}

moduleinterface IBidirectionalChannel
{
    gates:
        inout a;
        inout b;
}

moduleinterface IUnidirectionalChannel
{
    gates:
        input i;
        output o;
}
```

A.3 Packages

NED supports hierarchical namespaces called *packages*. The model is similar to Java packages, with minor changes.

A.3.1 Package Declaration

A NED file may contain a package declaration. The package declaration uses the **package** keyword, and specifies the package for the definitions in the NED file. If there is no package declaration, the file's contents are in the *default package*.

Component type names must be unique within their package.

A.3.2 Directory Structure, package.ned

Like in Java, the directory of a NED file must match the package declaration. However, it is possible to omit directories at the top which do not contain any NED files (like the typical `/org/<projectname>` directories in Java).

The top of a directory tree containing NED files is named a *NED source folder*.

> **NOTE:** The OMNEST runtime recognizes a `NEDPATH` environment variable, which contains a list of NED source folders, and is similar to the Java `CLASSPATH` variable. `NEDPATH` also has a command-line option equivalent.

The `package.ned` file at the top level of a NED source folder plays a special role.

If there is no toplevel `package.ned` or it contains no package declaration, the declared package of a NED file in the folder *<srcfolder>*`/x/y/z` *must* be `x.y.z`. If there is a toplevel `package.ned` and it declares the package as `a.b`, then any NED file in the folder *<srcfolder>*`/x/y/z` *must* have the declared package `a.b.x.y.z`.

> **NOTE:** `package.ned` files are allowed in other folders as well. They may contain properties and/or documentation for their package, but cannot be used to define the package they are in.

A.4 Components

Simple modules, compound modules, networks, channels, module interfaces and channel interfaces are called *components*.

A.4.1 Simple Modules

Simple module types are declared with the **`simple`** keyword; see the NED Grammar (Appendix B) for the syntax.

Simple modules may have properties (A.4.8), parameters (A.4.9) and gates (A.4.11).

A simple module type may not have inner types (A.4.15).

A simple module type may extend another simple module type, and may implement one or more module interfaces (A.4.5). Inheritance rules are described in section A.4.21, and interface implementation rules in section A.4.20.

Every simple module type has an associated C++ class, which must be subclassed from `cSimpleModule`. The way of associating the NED type with the C++ class is described in section A.4.7.

A.4.2 Compound Modules

Compound module types are declared with the **`module`** keyword; see the NED Grammar (Appendix B) for the syntax.

A compound module may have properties (A.4.8), parameters (A.4.9), and gates (A.4.11); its internal structure is defined by its submodules (A.4.12) and connections (A.4.13); and it may also have inner types (A.4.15) that can be used for its submodules and connections.

A compound module type may extend another compound module type, and may implement one or more module interfaces (A.4.5). Inheritance rules are described in section A.4.21, and interface implementation rules in section A.4.20.

A.4.3 Networks

The network Keyword

A network declared with the **`network`** keyword is equivalent to a compound module (**`module`** keyword) with the `@isNetwork(true)` property.

NOTE: A simple module can only be designated to be a network by spelling out the `@isNetwork` property; the **`network`** keyword cannot be used for that purpose.

The @isNetwork Property

The `@isNetwork` property is only recognized for simple modules and compound modules. The value may be empty, true or false:

```
@isNetwork;
@isNetwork();
@isNetwork(true);
@isNetwork(false);
```

The empty value corresponds to `@isNetwork(true)`.

The `@isNetwork` property is not inherited; that is, a subclass of a module with `@isNetwork` set does not automatically become a network. The `@isNetwork` property needs to be explicitly added to the subclass to make it a network.

Rationale: Subclassing may introduce changes to a module that make it unfit to be used as a network.

A.4.4 Channels

Channel types are declared with the **`channel`** keyword; see the NED Grammar (Appendix B) for the syntax.

Channel types may have properties (A.4.8) and parameters (A.4.9).

A channel type may not have inner types (A.4.15).

A channel type may extend another channel type, and may implement one or more channel interfaces (A.4.6). Inheritance rules are described in section A.4.21, and interface implementation rules in section A.4.20.

Every channel type has an associated C++ class, which must be subclassed from `cChannel`. The way of associating the NED type with the C++ class is described in section A.4.7.

The `@defaultname` property of a channel type determines the default name of the channel object when used in a connection.

A.4.5 Module Interfaces

Module interface types are declared with the **`moduleinterface`** keyword; see the NED Grammar (Appendix B) for the syntax.

Module interfaces may have properties (A.4.8), parameters (A.4.9), and gates (A.4.11). However, parameters are not allowed to have a value assigned, not even a default value.

A module interface type may not have inner types (A.4.15).

A module interface type may extend one or more other module interface types. Inheritance rules are described in section A.4.21.

A.4.6 Channel Interfaces

Channel interface types are declared with the **channelinterface** keyword; see the NED Grammar (Appendix B) for the syntax.

Channel interfaces may have properties (A.4.8) and parameters (A.4.9). However, parameters are not allowed to have a value assigned, not even a default value.

A channel interface type may not have inner types (A.4.15).

A channel interface type may extend one or more other channel interface types. Inheritance rules are described in section A.4.21.

A.4.7 Resolving the C++ Implementation Class

The procedure for determining the C++ implementation class for simple modules and for channels are identical. It goes as follows (we are going to say *component* instead of *"simple module or channel"*):

If the component extends another component and has no `@class` property, the C++ implementation class is inherited from the base type.

If the component contains a `@class` property, the C++ class name will be composed of the *current namespace* (see below) and the value of the `@class` property. The `@class` property should contain a single value.

> **NOTE:** The `@class` property may itself contain a namespace declaration (ie. may contain "`::`").

If the component contains no `@class` property and has no base class, the C++ class name will be composed of the *current namespace* and the unqualified name of the component.

> **IMPORTANT:** Subclassing in NED does not imply subclassing the C++ implementation. If one intends to subclass a simple module or channel in NED as well as in C++, the `@class` property needs to be explicitly specified in the derived type, otherwise it will continue to use the C++ class from its super type.

Compound modules will be instantiated with the built-in `cModule` class, unless the module contains the `@class` property. When `@class` is present, the resolution rules are the same as with simple modules.

Current Namespace

The *current namespace* is the value of the first `@namespace` property found while searching the following order:

1. the current NED file
2. the `package.ned` file in the current package or the first ancestor package searching upwards

> **NOTE:** Note that namespaces coming from multiple `@namespace` properties in different scopes do not nest, but rather, the nearest one wins.

The `@namespace` property should contain a single value.

A.4.8 Properties

Properties are a means of adding metadata annotations to NED files, component types, parameters, gates, submodules, and connections.

Identifying a Property

Properties are identified by name. It is possible to have several properties on the same object with the same name, as long as they have unique indices. An index is an identifier in square brackets after the property name.

The following example shows a property without index, one with the index `index1`, and a third with the index `index2`.

```
@prop();
@prop[index1]();
@prop[index2]();
```

Property Value

The value of the property is specified inside parentheses. The property value consists of *key=valuelist* pairs, separated by semicolons; *valuelist* elements are separated with commas. Example:

```
@prop(key1=value11,value12,value13;key2=value21,value22)
```

Keys must be unique.

If the key+equal sign part (`key=`) is missing, the valuelist belongs to the *default key*. Examples:

```
@prop1(value1,value2)
@prop2(value1,value2;key1=value11,value12,value13)
```

Most of the properties use the default key with one value. Examples:

```
@namespace(inet);
@class(Foo);
@unit(s);
```

Property values have a liberal syntax (see Appendix B). Values that do not fit the grammar (notably, those containing a comma or a semicolon) need to be surrounded with double quotes.

When interpreting a property value, one layer of quotes is removed automatically, that is, `foo` and `"foo"` are the same. Within quotes, escaping works in the same way as within string literals (see A.1.6).

Example:

```
@prop(marks=the ! mark, "the , mark", "the ; mark", other marks); // 4 items
```

Placement

Properties may be added to NED files, component types, parameters, gates, submodules and connections. For the exact syntax, see Appendix B.

When a component type extends another component type(s), properties are merged. This is described in section A.4.21.

Property Declarations

The **property** keyword is reserved for future use. It is envisioned that accepted property names and property keys would need to be pre-declared, so that the NED infrastructure can warn the user about mistyped or unrecognized names.

A.4.9 Parameters

Parameters can be defined and assigned in the **parameters** section of component types. In addition, parameters can also be assigned in the **parameters** sections of submodule bodies and connection bodies, but those places do not allow adding new parameters.

The **parameters** keyword is optional, and can be omitted without change in the meaning.

The **parameters** section may also hold pattern assignments (A.4.10) and properties (A.4.8).

A parameter is identified by a name, and has a data type. A parameter may have value or default value, and may also have properties (see A.4.8).

Accepted parameter data types are **double**, **int**, **string**, **bool**, **xml**, and **object**. Any of the above types can be declared **volatile** as well (`volatile int`, `volatile string`, etc.)

The presence of a data type keyword determines whether the given line defines a new parameter or refers to an existing parameter. One can assign a value or default value to an existing parameter, and/or modify its properties or add new properties.

Examples:

```
int a;            // defines new parameter
int b @foo;       // new parameter with property
int c = default(5); // new parameter with default value
int d = 5;        // new parameter with value assigned
int e @foo = 5;   // new parameter with property and value
f = 10;           // assignment to existing (e.g.inherited) parameter
g = default(10); // overrides default value of existing parameter
h;                // legal, but does nothing
i @foo(1);        // adds a property to existing parameter
j @foo(1) = 10;   // adds a property and value to existing parameter
```

Parameter values are NED expressions. Expressions are described in section A.5.

For **volatile** parameters, the value expression is evaluated every time the parameter value is accessed. Non-**volatile** parameters are evaluated only once.

> **NOTE:** The **const** keyword is reserved for future use within expressions to define constant subexpressions, i.e. to denote a part within an expression that should only be evaluated once. Constant subexpressions are not supported yet.

The following properties are recognized for parameters: `@unit`, `@prompt`, `@mutable`.

The @prompt Property

The `@prompt` property defines a prompt string for the parameter. The prompt string is used when/if a simulation runtime user interface interactively prompts the user for the parameter's value.

The `@prompt` property is expected to contain one string value for the default key.

The @unit Property

A parameter may have a `@unit` property to associate it with a measurement unit. The `@unit` property should contain one string value for the default key. Examples:

```
@unit(s)
@unit(second)
```

When present, values assigned to the parameter must be in the same or in a compatible (that is, convertible) unit. Examples:

```
  double a @unit(s) = 5s;    // OK
  double a @unit(s) = 10ms; // OK; will be converted to seconds
  double a @unit(s) = 5;     // error: should be 5s
  double a @unit(s) = 5kg;   // error: incompatible unit
```

`@unit` behavior for non-numeric parameters (boolean, string, XML) is unspecified (may be ignored or may be an error).

The `@unit` property of a parameter may not be modified via inheritance.

Example:

```
simple A {
    double p @unit(s);
}
simple B extends A {
    p @unit(mW);  // illegal: cannot override @unit
}
```

The @mutable Property

When a parameter is annotated with `@mutable`, the parameter's value is allowed to be changed at runtime, i.e. after its module has been set up. Parameters without the `@mutable` property cannot be changed at runtime.

A.4.10 Pattern Assignments

Pattern assignments allow one to set more than one parameter using wildcards, and to assign parameters deeper down in a submodule tree. Pattern assignments may occur in the **`parameters`** section of component types, submodules and connections.

The syntax of a pattern assignment is <*pattern*> = <*value*>.

A pattern consists of two or more pattern elements, separated by dots. The pattern element syntax is defined so that it can accommodate names of parameters, submodules (optionally with index), gates (optionally with the `$i`/`$o` suffix and/or index) and connections, and their wildcard forms. (The default name of connection channel objects is **`channel`**.)

Wildcard forms may use:

1. Asterisks: They match zero or more characters except dots.
2. Numeric ranges, `{`*<start>*`..`*<end>*`}` e.g. `{5..120}` or `{..10}`. They match numbers embedded in identifiers, that is, a sequence of decimal digit characters interpreted as a nonnegative integer that is within the specified *start..end* range (both limits are inclusive). Both *start* and *end* are optional.
3. Numeric index ranges, `[`*<start>*`..`*<end>*`]`. e.g. `[5..120]` or `[..10]`. They are intended for selecting submodule and gate index ranges. They match a nonnegative integer enclosed in square brackets that is within the specified *start..end* range (both limits are inclusive). Both *start* and *end* are optional.
4. Double asterisks: They match zero or more characters (including dots), and can be used to match more than one parameter path elements.

See the NED language grammar (Appendix B) for a more formal definition of the pattern syntax.

Examples:

```
host1.tcp.mss = 512B;
host*.tcp.mss = 512B; // matches host, host1, host2, hostileHost, ...
host{9..11}.tcp.mss = 512B; // matches host9/host10/host11, but nothing else
host[9..11].tcp.mss = 512B; // matches host[9]/host[10]/host[11], but nothing else
**.mss = 512B; // matches foo.mss, host[1].transport.tcp[0].mss, ...
```

A.4.11 Gates

Gates can be defined in the **`gates`** section of component types. The size of a gate vector (see below) may be specified at the place of defining the gate, via inheritance in a derived type, and also in the **`gates`** block of a submodule body. A submodule body does not allow defining new gates.

A gate is identified by a name, and is characterized by a type (**`input`**, **`output`**, **`inout`**) and optionally a vector size. Gates may also have properties (see A.4.8).

Gates may be scalar or vector. The vector size is specified with a numeric expression inside square brackets. The vector size may also be left unspecified by writing an empty pair of square brackets.

An already specified gate vector size may not be overridden in subclasses or in a submodule.

The presence of a gate type keyword determines whether the given line defines a new gate or refers to an existing gate. One can specify the gate vector size for an existing gate vector, and/or modify its properties, or add new properties.

Examples:

```
gates:
  input a;          // defines new gate
```

```
input b @foo;     // new gate with property
input c[];        // new gate vector with unspecified size
input d[8];       // new gate vector with size=8
e[10];            // set gate size for existing (e.g.inherited) gate vector
f @foo(bar);      // add property to existing gate
g[10] @foo(bar); // set gate size and add property to existing gate
```

Gate vector sizes are NED expressions. Expressions are described in section A.5.

See the Connections section (A.4.13) for more information on gates.

Recognized Gate Properties

The following properties are recognized for gates: `@directIn` and `@loose`. They have the same effect: When either of them is present on a gate, the gate is not required to be connected in the connections section of a compound module (see A.4.13).

`@directIn` should be used when the gate is an **`input`** gate that is intended for being used as a target for the `sendDirect()` method; `@loose` should be used in any other case when the gate is not required to be connected for some reason.

> **NOTE:** The reason `@directIn` gates are not *required* to remain unconnected is that it is often useful to wrap such modules in a compound module, where the compound module also has a `@directIn` input gate that is internally connected to the submodule's corresponding gate.

Example:

```
gates:
    input radioIn @directIn;
```

A.4.12 Submodules

Submodules are defined in the **`submodules`** section of the compound module.

The type of the submodule may be specified statically or parametrically.

Submodules may be scalar or vector. The size of submodule vectors must be specified as a numeric expression inside square brackets.

Submodules may also be conditional.

A submodule definition may or may not have a body (a curly brace delimited block). An empty submodule body is equivalent to a missing one.

Syntax examples:

```
submodules:
    ip : IP;          // scalar submodule without body
    tcp : TCP {}      // scalar submodule with empty body
    app[10] : App;    // submodule vector
```

Submodule Type

The simple or compound module type (A.4.1, A.4.2) that will be instantiated as the submodule may be specified either statically (with a concrete module type name) or parametrically.

Static Submodule Type

Submodules with a statically defined type are those that contain a concrete NED module type name. Example:

```
tcp : TCP;
```

See section A.4.18 for the type resolution rules.

Parametric Submodule Type

Parametric submodule type means that the NED type name is given in a string expression. The string expression may be specified locally in the submodule declaration, or elsewhere using typename patterns (see later).

Parametric submodule types are syntactically denoted by the presence of an expression in a pair of angle brackets and the **like** keyword followed by a module interface type A.4.5 that a module type must implement in order to be eligible to be chosen. The angle brackets may be empty, contain a string expression, or contain a default string expression (`default(...)` syntax).

Examples:

```
tcp : <tcpType> like ITCP;         // type comes from parent module parameter
tcp : <"TCP_"+suffix> like ITCP;   // expression using parent module parameter

tcp : <> like ITCP;                // type must be specified elsewhere

tcp : <default("TCP")> like ITCP; // type may be specified elsewhere;
                                   // if not, the default is "TCP"
tcp : <default("TCP_"+suffix)> like ITCP;
                                   // type may be specified elsewhere;
                                   // if not, the default is an expression
```

See the NED Grammar (Appendix B) for the formal syntax, and section A.4.19 for the type resolution rules.

Conditional Submodules

Submodules may be made conditional using the **if** keyword. The condition expression must evaluate to a boolean; if the result is `false`, the submodule is not created, and trying to connect its gates or reference its parameters will be an error.

An example:

```
submodules:
  tcp : TCP if withTCP { ... }
```

Parameters, Gates

A submodule body may contain parameters (A.4.9) and gates (A.4.5).

A submodule body cannot define new parameters or gates. It is only allowed to assign existing parameters, and to set the vector size of existing gate vectors.

It is also allowed to add or modify submodule properties and parameter/gate properties.

A.4.13 Connections

Connections are defined in the **`connections`** section of the compound module.

Connections may not span multiple hierarchy levels, that is, a connection may be created between two submodules, a submodule and the compound module, or between two gates of the compound module.

Normally, all gates must be connected, including submodule gates and the gates of the compound module. When the **`allowunconnected`** modifier is present after **`connections`**, gates will be allowed to be left unconnected.

> **NOTE:** The `@directIn` and `@loose` gate properties are alternatives to the `connections allowunconnected` syntax; see A.4.11.

Connections may be conditional, and may be created using loops (see A.4.14).

Connection Syntax

The connection syntax uses arrows (`-->`, `<--`) to connect **`input`** and **`output`** gates, and double arrows (`<-->`) to connect **`inout`** gates. The latter is also said to be a bidirectional connection.

Arrows point from the source gate (a submodule output gate or a compound module input gate) to the destination gate (a submodule input gate or a compound module output gate). Connections may be written either left to right or right to left, that is, `a-->b` is equivalent to `b<--a`.

Gates are specified as *<modulespec>.<gatespec>* (to connect a submodule), or as *<gatespec>* (to connect the compound module). *<modulespec>* is either a submodule name (for scalar submodules), or a submodule name plus an index in square brackets (for submodule vectors). For scalar gates, *<gatespec>* is the gate name; for gate vectors it is either the gate name plus a numeric index expression in square brackets, or *<gatename>*`++`.

The *<gatename>*`++` notation causes the first unconnected gate index to be used. If all gates of the given gate vector are connected, the behavior is different for submodules and for the enclosing compound module. For submodules, the gate vector expands by one. For the compound module, it is an error to use `++` on a gate vector with no unconnected gates.

Syntax examples:

```
connections:
   a.out --> b.in;   // unidirectional between two submodules
   c.in[2] <-- in;   // parent-to-child; gate vector with index
   d.g++ <--> e.g++; // bidirectional, auto-expanding gate vectors
```

Rationale: The reason it is not supported to expand the gate vector of the compound module is that the module structure is built in top-down order: new gates would be left unconnected on the outside, as there is no way in NED to "go back" and connect them afterwards.

When the `++` operator is used with `$i` or `$o` (e.g. `g$i++` or `g$o++`, see later), it will actually add a gate pair (input+output) to maintain equal gate size for the two directions.

The syntax to associate a channel (see A.4.4) with the connection is to use two arrows with a channel specification in between (see later). The same syntax is used to add properties such as `@display` to the connection.

Inout Gates

An inout gate is represented as a gate pair: an input gate and an output gate. The two subgates may also be referenced and connected individually, by adding the `$i` and `$o` suffix to the name of the inout gate.

A bidirectional connection (which uses a double arrow to connect two inout gates), is also a shorthand for two uni-directional connections; that is,

```
a.g <--> b.g;
```

is equivalent to

```
a.g$o --> b.g$i;
a.g$i <-- b.g$o;
```

In inout gate vectors, gates are always in pairs, that is, `sizeof(g$i)==sizeof(g$o)` always holds. It is maintained even when `g$i++` or `g$o++` is used: the `++` operator will add a gate pair, not just an input or an output gate.

Specifying Channels

A channel specification associates a channel object with the connection. A channel object is an instance of a channel type (see A.4.4).

NOTE: As bidirectional connections are a shorthand for a pair of uni-directional connections, they will actually create *two* channel objects, one for each direction.

The channel type to be instantiated may be implicit, or may be specified statically or parametrically.

A connection may have a body (a curly brace delimited block) for setting properties and/or parameters of the channel.

A connection syntax allows one to specify a name for the channel object. When not specified, the channel name will be taken from the `@defaultname` property of the channel type; when there is no such property, it will be `"channel"`. Custom connection names can be useful for easier addressing of channel objects when assigning parameters using patterns.

See subsequent sections for details.

Implicit Channel Type

If the connection syntax does not say anything about the channel type, it is implicitly determined from the set of connection parameters used.

Syntax examples for connections with implicit channel types:

```
a.g <--> b.g; // no parameters
a.g <--> {delay = 1ms;} <--> b.g;  // assigns delay
a.g <--> {datarate = 100Mbps; delay = 50ns;} <--> b.g; // assigns delay and datara
```

For such connections, the actual NED type to be used will depend on the parameters set in the connection:

1. When no parameters are set, `ned.IdealChannel` is chosen.
2. When only `ned.DelayChannel` parameters are used (`delay` and `disabled`), `ned.DelayChannel` is chosen.
3. When only `ned.DatarateChannel` parameters are used (`datarate`, `delay`, `ber`, `per`, `disabled`), the chosen channel type will be `ned.DatarateChannel`.

Connections with implicit channel types may not use any other parameter.

Static Channel Type

Connections with a statically defined channel type are those that contain a concrete NED channel type name.

Examples:

```
a.g <--> FastEthernet <--> b.g;
a.g <--> FastEthernet {per = 1e-6;} <--> b.g;
```

See section A.4.18 for the type resolution rules.

Parametric Channel Type

Parametric channel types are similar to parametric submodule types, described in section A.4.12.

Parametric channel type means that the NED type name is given in a string expression. The string expression may be specified locally in the connection declaration, or elsewhere using typename patterns (see later).

Parametric channel types are syntactically denoted by the presence of an expression in a pair of angle brackets and the **`like`** keyword followed by a channel interface type A.4.6 that a channel type must implement in order to be eligible to be chosen. The angle brackets may be empty, contain a string expression, or contain a default string expression (`default(...)` syntax).

Examples:

```
a.g++ <--> <channelType> like IMyChannel <--> b.g++;
                                    // type comes from parent module parameter
a.g++ <--> <"Ch_"+suffix> like IMyChannel <--> b.g++;
```

```
                                    // expression using parent module parameter
a.g++ <--> <> like IMyChannel <--> b.g++;
                                    // type must be specified elsewhere
a.g++ <--> <default("MyChannel")> like IMyChannel <--> b.g++;
                                    // type may be specified elsewhere;
                                    // if not, the default is "MyChannel"
a.g++ <--> <default("Ch_"+suffix)> like IMyChannel <--> b.g++;
                                    // type may be specified elsewhere;
                                    // if not, the default is an expression
```

See the NED Grammar (Appendix B) for the formal syntax, and section A.4.19 for the type resolution rules.

Channel Parameters and Properties

A channel definition may or may not have a body (a curly brace delimited block). An empty channel body (`{ }`) is equivalent to a missing one.

A channel body may contain parameters (A.4.9).

A channel body cannot define new parameters. It is only allowed to assign existing parameters.

It is also allowed to add or modify properties and parameter properties.

A.4.14 Conditional and Loop Connections, Connection Groups

The connections section may contain any number of connections and connection groups. A connection group is one or more connections grouped with curly braces.

Both connections and connection groups may be conditional (**`if`** keyword) or may be multiple (**`for`** keyword).

Any number of **`for`** and **`if`** clauses may be added to a connection or connection loop; they are interpreted as if they were nested in the given order. Loop variables of a **`for`** may be referenced from subsequent conditions and loops as well as in module and gate index expressions in the connections.

See the NED Grammar (B) for the exact syntax.

Example connections:

```
a.out --> b.in;
c.out --> d.in if p>0;
e.out[i] --> f[i].in for i=0..sizeof(f)-1, if i%2==0;
```

Example connection groups:

```
if p>0 {
    a.out --> b.in;
    a.in <-- b.out;
}
for i=0..sizeof(c)-1, if i%2==0 {
    c[i].out --> out[i];
    c[i].in <-- in[i];
}
```

```
for i=0..sizeof(d)-1, for j=0..sizeof(d)-1, if i!=j {
    d[i].out[j] --> d[j].in[i];
}
for i=0..sizeof(e)-1, for j=0..sizeof(e)-1 {
    e[i].out[j] --> e[j].in[i] if i!=j;
}
```

A.4.15 Inner Types

Inner types can be defined in the **types** section of compound modules, with the same syntax as toplevel (i.e. non-inner) types.

Inner types may not contain further inner types, that is, type nesting is limited to two levels.

Inner types are only visible inside the enclosing component type and its subclasses.

A.4.16 Name Uniqueness

Identifier names within a component must be unique. That is, the following items in a component are considered to be in the same name space and must not have colliding names:

- parameters
- gates
- submodules
- inner types
- the above items of super type(s)

For example, a gate and a submodule cannot have the same name.

A.4.17 Parameter Assignment Order

A module or channel parameter may be assigned in **parameters** blocks (see A.4.9) at various places in NED: in the module or channel type that defines it; in the type's subclasses; in the submodule or connection that instantiates the type. The parameter may also be assigned using pattern assignments (see A.4.10) in any compound module that uses the given module or channel type directly or indirectly.

Patterns are matched against the relative path of the parameter, which is the relative path of its submodule or connection, with a dot and the parameter name appended. The relative path is composed of a list of submodule names (name plus index) separated by dots; a connection is identified by the full name of its source gate plus the name of the channel object (which is currently always `channel`) separated by a dot.

> **NOTE:** As bidirectional connections are a shorthand for two unidirectional connections, the source gate name is qualified with `$i` or `$o` in the relative path.

Note that the **parameters** keyword itself is optional, and is usually not written out in submodules and connections.

This section describes the module and channel parameter assignments procedure.

The general rules are the following:

1. A (non-default) parameter assignment may not be overridden later; that is, if there are assignments in multiple places, the assignment "closest" to the parameter declaration will be effective; others will be flagged as errors.
2. A default value is only used if a non-default value is not present for the given parameter. A non-default value may also come from a source external to NED, namely the simulation configuration (`omnetpp.ini`).
3. Unlike non-default values, a default value *may* be overridden; that is, if there are default value assignments in multiple places, the assignment "farthest" from the parameter declaration will win.
4. Among pattern assignments within the same **parameters** block, the first match will win. Pattern assignments with default and non-default values are considered to be two disjoint sets, only one of which are searched at a time.

This yields the following conceptual search order for non-default parameter assignments:

1. First, the NED type that contains the parameter declaration is checked;
2. Then its subclasses are checked;
3. Then the submodule or connection that instantiates the type is checked;
4. Then the compound module that contains the submodule or connection is checked for matching pattern assignments;
5. Then, assuming the compound module is part of a network, the search for matching pattern assignments continues up on the module tree until the root (the module that represents the network). At each level (compound module), first the specific submodule definition is checked, then the (parent) compound module. If a compound module is subclassed before instantiated, the base type is checked first.

When no (non-default) assignment is found, the same places are searched in the *reverse order* for default value assignments. If no default value is found, an error may be raised or the user may be interactively prompted.

To illustrate the above rules, consider the following example where we want to assign parameter *p*:

```
simple A { double p; }
simple A2 extends A {...}
module B { submodules: a2: A2 {...} }
module B2 extends B {...}
network C { submodules: b2: B2 {...} }
```

Here, the search order is: *A, A2, a2, B, B2, b2, C*. NED conceptually searches the **parameters** blocks in that order for a (non-default) value, and then in reverse order for a default value.

The full search order and the form of assignment expected on each level:

1. `A { p = ...; }`
2. `A2 { p = ...; }`
3. `a2 { p = ...; }`
4. `B { a2.p = ...; }`
5. `B2 { a2.p = ...; }`
6. `b2 { a2.p = ...; }`
7. `C { b2.a2.p = ...; }`
8. `C { b2.a2.p = default(...); }`
9. `b2 { a2.p = default(...); }`
10. `B2 { a2.p = default(...); }`
11. `B { a2.p = default(...); }`
12. `a2 { p = default(...); }`
13. `A2 { p = default(...); }`
14. `A { p = default(...); }`

If only a default value is found or not even that, external configuration has a say. The configuration may contain an assignment for `C.b2.a2.p`; it may apply the default if there is one; it may ask the user interactively to enter a value; or if there is no default, it may raise an error *"no value for parameter"*.

A.4.18 Type Name Resolution

Names from other NED files can be referred to either by fully qualified name ("`inet.networklayer.ip.RoutingTable`"), or by short name ("`RoutingTable`") if the name is visible.

Visible names are:

- inner types of the same type or its super types;
- anything from the same package;
- imported names.

Imports

Imports have a similar syntax to Java, but they are more flexible with wildcards. All of the following are legal:

```
import inet.networklayer.ipv4.RoutingTable;
import inet.networklayer.ipv4.*;
import inet.networklayer.ipv4.Ro*Ta*;
import inet.*.ipv4.*;
import inet.**.RoutingTable;
```

One asterisk stands for any character sequence not containing dots; and a double asterisk stands for any character sequence (which may contain dots). No other wildcards are recognized.

An import not containing a wildcard must match an existing NED type. However, it is legal for an import that does contain wildcards not to match any NED type (although that might generate a warning.)

Inner types may not be referenced outside their enclosing types and their subclasses.

Base Types and Submodules

Fully qualified names and simple names are accepted. Simple names are looked up among the inner types of the enclosing type (compound module), then using imports, then in the same package.

Network Name in the Ini File

The network name in the ini file may be given as a fully qualified name or as a simple (unqualified) name.

Simple (unqualified) names are tried with the same package as the ini file is in (provided it is in a NED directory).

A.4.19 Resolution of Parametric Types

This section describes the type resolution for submodules and connections that are defined using the **like** keyword.

Type resolution is done in two steps. In the first step, the type name string expression is found and evaluated. Then in the second step, the resulting type name string is resolved to an actual NED type.

Step 1. The lookup of the type name string expression is similar to that of a parameter value lookup (A.4.17).

The expression may be specified locally (between the angle brackets), or using typename pattern assignments in any compound module that contains the submodule or connection directly or indirectly. A typename pattern is a pattern that ends in `.typename`.

Patterns are matched against the relative path of the submodule or connection, with `.typename` appended. The relative path is composed of a list of submodule names (name plus index) separated by dots; a connection is identified by the full name of its source gate plus the name of the channel object (which is currently always `channel`) separated by a dot.

> **NOTE:** As bidirectional connections are a shorthand for two unidirectional connections, the source gate name is qualified with `$i` or `$o` in the relative path.

An example that uses typename pattern assignment:

```
module Host {
    submodules:
        tcp: <> like ITCP;;
        ...
```

```
    connections:
        tcp.ipOut --> <> like IMyChannel --> ip.tcpIn;
}

network Network {
    parameters:
        host[*].tcp.typename = "TCP_lwIP";
        host[*].tcp.ipOut.channel.typename = "DebugChannel";
    submodules:
        host[10] : Host;
        ...
}
```

The general rules are the following:

1. A (non-default) parameter assignment may not be overridden later; that is, if there are assignments in multiple places, the assignment "closest" to the submodule or connection definition will be effective; others will be flagged as errors.
2. A default value is only used if a non-default value is not present. A non-default value may also come from a source external to NED, namely the simulation configuration (`omnetpp.ini`).
3. Unlike non-default values, a default value *may* be overridden; that is, if there are default value assignments in multiple places, the assignment "farthest" from the submodule or connection definition will win.
4. Among pattern assignments within the same **`parameters`** block, the first match will win. Patterns assignments with default and non-default values are considered to be two disjoint sets, only one of which are searched at a time.

This yields the following conceptual search order for typename assignments:

1. First, the submodule or connection definition is checked (angle brackets);
2. Then the compound module that contains the submodule or connection is checked for matching pattern assignments;
3. Then, assuming the compound module is part of a network, the search for matching pattern assignments continues up on the module tree until the root (the module that represents the network). At each level (compound module), first the specific submodule definition is checked, then the (parent) compound module. If a compound module is subclassed before instantiated, the base type is checked first.

When no (non-default) assignment is found, the same places are searched in the *reverse order* for default value assignments. If no default value is found, an error may be raised or the user may be interactively prompted.

To illustrate the above rules, consider the following example:

```
module A { submodules: h: <> like IFoo; }
module A2 extends A {...}
module B { submodules: a2: A2 {...} }
module B2 extends B {...}
network C { submodules: b2: B2 {...} }
```

Here, the search order is: *h, A, A2, a2, B, B2, b2, C*. NED conceptually searches the **parameters** blocks in that order for a (non-default) value, and then in reverse order for a default value.

The full search order and the form of assignment expected on each level:

1. `h: <...> like IFoo;`
2. `A { h.typename = ...; }`
3. `A2 { h.typename = ...; }`
4. `a2 { h.typename = ...; }`
5. `B { a2.h.typename = ...; }`
6. `B2 { a2.h.typename = ...; }`
7. `b2 { a2.h.typename = ...; }`
8. `C { b2.a2.h.typename = ...; }`
9. `C { b2.a2.h.typename = default(...); }`
10. `b2 { a2.h.typename = default(...); }`
11. `B2 { a2.h.typename = default(...); }`
12. `B { a2.h.typename = default(...); }`
13. `a2 { h.typename = default(...); }`
14. `A2 { h.typename = default(...); }`
15. `A { h.typename = default(...); }`
16. `h: <default(...)> like IFoo;`

If only a default value is found or not even that, external configuration has a say. The configuration may contain an assignment for `C.b2.a2.h.typename`; it may apply the default value if there is one; it may ask the user interactively to enter a value; or if there is no default value, it may raise an error *"cannot determine submodule type"*.

Step 2. The type name string is expected to hold the simple name or fully qualified name of the desired NED type. Resolving the type name string to an actual NED type differs from normal type name lookups in that it ignores the imports in the file altogether. Instead, a list of NED types that have the given simple name or fully qualified name *and* implement the given interface is collected. The result must be exactly one module or channel type.

A.4.20 Implementing an Interface

A module type may implement one or more module interfaces, and a channel type may implement one or more channel interfaces, using the **like** keyword.

The module or channel type is required to have *at least* those parameters and gates that the interface has.

Regarding component properties, parameter properties and gate properties defined in the interface: the module or channel type is required to have at least the properties of the interface, with at least the same values. The component may have additional properties, and properties may add more keys and values.

NOTE: Implementing an interface does not cause the properties, parameters and gates to be interited by the module or channel type; they have to be added explicitly.

NOTE: A module or channel type may have extra properties, parameters and gates in addition to those in the interface.

A.4.21 Inheritance

Component inheritance is governed by the following rules:

- A simple module may only extend a simple module.
- A compound module may only extend a compound module or a simple module.
- A channel may only extend a channel.
- A module interface may only extend a module interface (or several module interfaces).
- A channel interface may only extend a channel interface (or several channel interfaces).

A network is a shorthand for a compound module with the `@isNetwork` property set, so the same rules apply to it as to compound modules.

Inheritance may:

- add new properties, parameters, gates, inner types, submodules, connections, as long as names do not conflict with inherited names
- modify inherited properties, and properties of inherited parameters and gates
- it may not modify inherited submodules, connections and inner types

Other inheritance rules:

- for inner types: new inner types can be added, but inherited ones cannot be changed
- for properties: contents will be merged (rules like for display strings: values on same key and same position will overwrite old ones)
- for parameters: type cannot be redefined; value may be redefined in subclasses or at place of usage
- for gates: type cannot be redefined; vector size may be specified in subclasses or at place of usage if it was unspecified
- for gate/parameter properties: extra properties can be added; existing properties can be overridden/extended as for standalone properties
- for submodules: new submodules may be added, but inherited ones cannot be modified
- for connections: new connections may be added, but inherited ones cannot be modified

The following sections will elaborate on the above rules.

Property Inheritance

Generally, properties may be modified via inheritance. Inheritance may:

- add new keys
- add/overwrite values for existing keys
- remove a value from an existing key by using hyphen as a special value

Parameter Inheritance

Default values for parameters may be overridden in subclasses.

Gate Inheritance

Gate vector size may not be overridden in subclasses.

A.4.22 Network Build Order

When a network is instantiated for simulation, the module tree is built in a top-down preorder fashion. This means that starting from an empty system module, all submodules are created, their parameters and vector sizes are assigned, and they get fully connected before proceeding to go into the submodules to build their internals.

This implies that inside a compound module definition (including in submodules and connections), one can refer to the compound module's parameters and gate sizes, because they are already built at the time of usage.

The same rules apply to compound or simple modules created dynamically during runtime.

A.5 Expressions

NED language expressions have a C-like syntax, with some variations on operator names (see `^`, `#`, `##`). Expressions may refer to module parameters, loop variables (inside connection **`for`** loops), gate vector and module vector sizes, and other attributes of the model. Expressions can use built-in and user-defined functions as well. There is a JSON-like notation for defining arrays and (dictionary-like) objects.

> **NOTE:** New NED functions can be defined in C++; see section 7.12.

A.5.1 Constants

See section A.1.6.

A.5.2 Array and Object Values

A bracketed list of zero or more comma-separated expressions denotes an *array* value. Example: `[9.81,false,"Hello"]`.

A list of zero or more comma-separated key-value pairs enclosed in a pair of curly braces denotes an *object* value. A key and a value are separated by a colon. A key may be a name or a string literal. A value may be an arbitrary expression, including a list or an object. The open brace may be preceded by an (optionally namespace-qualified) *class name*. Example 1: `{name:"John", age: 31}`. Example 2: `Filter {dest:"10.0.0.1", port:1200}`.

Array and *object* values may be assigned to parameters of type **`object`**. Note that **`null`** / **`nullptr`** are also of type *object*.

Array values are represented with the C++ class `cValueArray`, and by default, *object* values with the C++ class `cValueMap`. If the *object* notation includes a *class name*, then the named C++ class will be used instead of `cValueMap`, and filled in using the key-value list with the help of the class descriptor (`cClassDescriptor`) of the class, interpreting keys as field names.

A.5.3 Operators

The following operators are supported (in order of decreasing precedence):

Operator	Meaning
`-`, `!`, `~`	unary minus, negation, bitwise complement
`^`	power-of
`*`, `/`, `%`	multiply, divide, integer modulo
`+`, `-`	add, subtract, string concatenation
«, »	bitwise shift
`&`	bitwise and
`#`	bitwise xor
`|`	bitwise or
`=~`	string match
`<=>`	three-way comparison, a.k.a. “spaceship operator”
`>`, `>=` `<`, `<=`	greater than, greater than or equal to less than, less than or equal to
`==` `!=`	equal not equal
`&&`	logical operator and
`##`	logical operator xor
`||`	logical operator or
`?:`	the C/C++ “inline if”

The spaceship operator is defined as follows. The result of `a <=> b` is negative if `a<b`, zero if `a==b`, and positive if `a>b`. If either `a` or `b` is `nan` (not-a-number), the result is `nan` as well.

The string match operator works the following way. `x =~ pattern` returns `true` if the string `x` matches the string `pattern`, and `false` otherwise. Pattern syntax and rules: case sensitive, full-string match, where an asterisk `*` matches zero or more of any character except dot; a double asterisk `**` matches zero or more characters (including dot); a pair of curly braces containing a numeric range matches an embedded whole number in that range; a pair of square brackets containing a numeric range matches a number in that range enclosed in square brackets. A numeric range has the syntax of *<start>..<end>*, where both *<start>* and

<end> are integers, and are optional.

The interpretation of other operators is similar to that in C/C++.

Conversions

Values may have the same types as NED parameters: boolean, integer, double, string, XML element, and object. An integer or double value may have an associated measurement unit (`s`, `mW`, etc.)

Double-to-integer conversions require explicit cast using the `int()` function, there is no implicit conversion.

Integer-to-double conversion is implicit. However, a runtime error will be raised if there is precision loss during the conversion, i.e. the integer is so large that it cannot be precisely represented in a double. That error can be suppressed by using an explicit cast (`double()`).

There is no implicit conversion between boolean and numeric types, so `0` is not a synonym for **`false`**, and nonzero numbers are not a synonym for **`true`**.

There is also no conversion between string and numeric types, so e.g. `"foo"+5` is illegal. There are functions for converting a number to string and vice versa.

Bitwise operators expect integer arguments.

> **NOTE:** Integers are represented with 64-bit signed integers (`int64_t` in C++).

Unit Handling

Operations involving numbers with units work in the following way:

Addition, subtraction, and numeric comparisons require their arguments to have the same unit or compatible units; in the latter case a unit conversion is performed before the operation. Incompatible units cause an error.

Modulo, power-of and the bitwise operations require their arguments to be dimensionless, otherwise the result would depend on the choice of the unit.

> **NOTE:** If one needs a floating-point modulo operator that handles units as well, the `fmod()` function can be used.

Multiplying two numbers with units is not supported.

For division, dividing two numbers with units is only supported if the two units are convertible (i.e. the result will be dimensionless). Dividing a dimensionless number with a number with unit is not supported.

Operations involving quantities with logarithmic units (dB, dBW, etc.) are not supported, except for comparisons. (The reason is that such operations would be easy to misinterpret. For example, it is not obvious whether `10dB+10dB` (3.16+3.16) should evaluate to 20dB (=10.0) or to 16.02dB (=2*3.16=6.32), considering that such quantities would often be hidden behind parameter names where the unit is not obvious.)

A.5.4 Referencing Parameters and Loop Variables

Identifiers in expressions occurring *anywhere* in component definitions are interpreted as referring to parameters of the given component. For example, identifiers inside submodule bodies refer to the parameters of the *compound* module.

Expressions may also refer to parameters of submodules defined earlier in the NED file, using the `submoduleName.paramName` or the `submoduleName[index].paramName` syntax. To refer to parameters of the local module/channel inside a submodule or channel body, use the **`this`** qualifier: `this.destAddress`. To make a reference to a parameter of the compound module from within a submodule or channel body explicit, use the **`parent`** qualifier: `parent.destAddress`. [1]

Exception: if an identifier occurs in a connection **`for`** loop and names a previously defined loop variable, then it is understood as referring to the loop variable.

A.5.5 The `typename` Operator

The **`typename`** operator returns the NED type name as a string. If it occurs inside a component definition but outside a submodule or channel block, it returns the type name of the component being defined. If it occurs inside a submodule or channel block, it returns the type name of that submodule or channel.

The **`typename`** may also occur in the **`if`** condition of a (scalar) submodule or connection. In such cases, it evaluates to the *would-be* type name of the submodule or condition. This allows for conditional instantiation of parametric-type submodules, controlled from a **`typename`** assignment. (For example, by using the `if typename!=""` condition, one allows the submodule to be omitted by configuring `typename=""` for it.

`typename` is not allowed in a submodule vector's **`if`** condition. The reason is that the condition applies to the vector as a whole, while type is per-element.

A.5.6 The `index` Operator

The **`index`** operator is only allowed in a vector submodule's body, and yields the index of the submodule instance.

A.5.7 The `exists()` Operator

The **`exists()`** operator takes one identifier as argument, and it is only accepted in compound module definitions. The identifier must name a previously defined submodule, which will typically be a conditional submodule. The operator returns **`true`** if given submodule exists (has been created), and **`false`** otherwise.

A.5.8 The `sizeof()` Operator

The **`sizeof()`** operator expects one argument, and it is only accepted in compound module definitions.

[1] The **`parent`** qualifier is available from OMNEST 5.7.

The `sizeof(identifier)` syntax occurring *anywhere* in a compound module yields the size of the named submodule or gate vector of the compound module.

Inside submodule bodies, the size of a gate vector of the same submodule can be referred to with the **`this`** qualifier: `sizeof(this.out)`.

To refer to the size of a submodule's gate vector defined earlier in the NED file, use the `sizeof(submoduleName.gateVectorName)` or `sizeof(submoduleName[index].gateVectorName)` syntax.

A.5.9 The `expr()` Operator

The **`expr()`** operator allows a mathematical formula or other expression to be passed to a component as an object. **`expr()`** expects an expression as argument, and returns an object which encapsulates the expression in a parsed form. In the intended use case, the returned *expression object* is assigned to a module parameter, and is later utilized by user code (a component implementation) by binding its free variables and evaluating it. Identifiers in the expression are *not* interpreted as parameter references as in NED, but as free variables.

A.5.10 Functions

The functions available in NED are listed in Appendix D.

Selected functions are documented below.

The `xmldoc()` Function

The **`xmldoc()`** NED function can be used to assign **`xml`** parameters, that is, point them to XML files or to specific elements inside XML files.

`xmldoc()` accepts a file name as well as an optional second string argument that contains an XPath-like expression.

The XPath expression is used to select an element within the document. If the expression matches several elements, the first element (in preorder depth-first traversal) will be selected. (This is unlike XPath, which selects all matching nodes.)

The expression syntax is the following:

- An expression consists of *path components* (or "steps") separated by "`/`" or "`//`".
- A path component can be an element tag name, "`*`", "`.`" or "`..`".
- "`/`" means child element (just as in `/usr/bin/gcc`); "`//`" means an element any number of levels under the current element.
- "`.`", "`..`" and "`*`" mean the current element, the parent element, and an element with any tag name, respectively.
- Element tag names and "`*`" can have an optional predicate in the form "`[position]`" or "`[@attribute='value']`". Positions start from zero.
- Predicates of the form "`[@attribute=`*`$param`*`]`" are also accepted, where *`$param`* can be one of: `$MODULE_FULLPATH`, `$MODULE_FULLNAME`, `$MODULE_NAME`, `$MODULE_INDEX`,

$MODULE_ID, $PARENTMODULE_FULLPATH, $PARENTMODULE_FULLNAME, $PARENTMODULE_NAME, $PARENTMODULE_INDEX, $PARENTMODULE_ID, $GRANDPARENTMODULE_FULLPATH, $GRANDPARENTMODULE_FULLNAME, $GRANDPARENTMODULE_NAME, $GRANDPARENTMODULE_INDEX, $GRANDPARENTMODULE_ID.

The `xml()` Function

The **`xml()`** NED function can be used to parse a string as an XML document, and assign the result to an **`xml`** parameter.

`xml()` accepts the string to be parsed as well as an optional second string argument that contains an XPath-like expression.

The XPath expression is used in the same manner as with the **`xmldoc()`** function.

A.5.11 Units of Measurement

The following measurements units are recognized in constants. Other units can be used as well, but there are no conversions available for them (i.e. `parsec` and `kiloparsec` will be treated as two completely unrelated units.)

Unit	Name	Value
d	day	86400s
h	hour	3600s
min	minute	60s
s	second	
ms	millisecond	0.001s
us	microsecond	1e-6s
ns	nanosecond	1e-9s
ps	picosecond	1e-12s
fs	femtosecond	1e-15s
as	attosecond	1e-18s
bps	bit/sec	
kbps	kilobit/sec	1000bps
Mbps	megabit/sec	1e6bps
Gbps	gigabit/sec	1e9bps
Tbps	terabit/sec	1e12bps
B	byte	8b
KiB	kibibyte	1024B
MiB	mebibyte	1.04858e6B
GiB	gibibyte	1.07374e9B
TiB	tebibyte	1.09951e12B
kB	kilobyte	1000B
MB	megabyte	1e6B
GB	gigabyte	1e9B
TB	terabyte	1e12B
b	bit	
Kib	kibibit	1024b
Mib	mebibit	1.04858e6b
Gib	gibibit	1.07374e9b

Tib	tebibit	1.09951e12b
kb	kilobit	1000b
Mb	megabit	1e6b
Gb	gigabit	1e9b
Tb	terabit	1e12b
rad	radian	
deg	degree	0.0174533rad
m	meter	
cm	centimeter	0.01m
mm	millimeter	0.001m
um	micrometer	1e-6m
nm	nanometer	1e-9m
km	kilometer	1000m
W	watt	
mW	milliwatt	0.001W
uW	microwatt	1e-6W
nW	nanowatt	1e-9W
pW	picowatt	1e-12W
fW	femtowatt	1e-15W
kW	kilowatt	1000W
MW	megawatt	1e6W
GW	gigawatt	1e9W
Hz	hertz	
kHz	kilohertz	1000Hz
MHz	megahertz	1e6Hz
GHz	gigahertz	1e9Hz
THz	terahertz	1e12Hz
kg	kilogram	
g	gram	0.001kg
K	kelvin	
J	joule	
kJ	kilojoule	1000J
MJ	megajoule	1e6J
Ws	watt-second	1J
Wh	watt-hour	3600J
kWh	kilowatt-hour	3.6e6J
MWh	megawatt-hour	3.6e9J
V	volt	
kV	kilovolt	1000V
mV	millivolt	0.001V
A	ampere	
mA	milliampere	0.001A
uA	microampere	1e-6A
Ohm	ohm	
mOhm	milliohm	0.001Ohm
kOhm	kiloohm	1000Ohm
MOhm	megaohm	1e6Ohm
mps	meter/sec	
kmps	kilometer/sec	1000mps

kmph	kilometer/hour	(1/3.6)mps
C	coulomb	1As
As	ampere-second	
mAs	milliampere-second	0.001As
Ah	ampere-hour	3600As
mAh	milliampere-hour	3.6As
ratio	ratio	
pct	percent	0.01ratio
dBW	decibel-watt	10*log10(W)
dBm	decibel-milliwatt	10*log10(mW)
dBmW	decibel-milliwatt	10*log10(mW)
dBV	decibel-volt	20*log10(V)
dBmV	decibel-millivolt	20*log10(mV)
dBA	decibel-ampere	20*log10(A)
dBmA	decibel-milliampere	20*log10(mA)
dB	decibel	20*log10(ratio)

Appendix B

NED Language Grammar

This appendix contains the grammar for the NED language.

In the NED language, space, horizontal tab and new line characters count as delimiters, so one or more of them is required between two elements of the description which would otherwise be unseparable.

'//' (two slashes) may be used to write comments that last to the end of the line.

The language is fully case sensitive.

Notation:

- rule syntax is that of *bison*
- uppercase words are terminals, lowercase words are nonterminals
- `NAME`, `STRINGCONSTANT`, `INTCONSTANT`, `REALCONSTANT` represent identifier names and string, integer and real number literals (defined as in the C language, except that a `0` prefix does not stand for octal notation)
- other terminals represent keywords in all lowercase

```
nedfile
        : definitions
        | %empty
        ;

definitions
        : definitions definition
        | definition
        ;

definition
        : packagedeclaration
        | import
        | propertydecl
        | fileproperty
        | channeldefinition
        | channelinterfacedefinition
```

```
        | simplemoduledefinition
        | compoundmoduledefinition
        | networkdefinition
        | moduleinterfacedefinition
        | ';'
        ;

packagedeclaration
        : PACKAGE dottedname ';'
        ;

dottedname
        : dottedname '.' NAME
        | NAME
        ;

import
        : IMPORT importspec ';'
        ;

importspec
        : importspec '.' importname
        | importname
        ;

importname
        : importname NAME
        | importname '*'
        | importname '**'
        | NAME
        | '*'
        | '**'
        ;

propertydecl
        : propertydecl_header opt_inline_properties ';'
        | propertydecl_header '(' opt_propertydecl_keys ')' opt_inline_properties ';'
        ;

propertydecl_header
        : PROPERTY '@' PROPNAME
        | PROPERTY '@' PROPNAME '[' ']'
        ;

opt_propertydecl_keys
        : propertydecl_keys
        | %empty
        ;

propertydecl_keys
        : propertydecl_keys ';' propertydecl_key
        | propertydecl_key
```

```
        ;

propertydecl_key
        : property_literal
        ;

fileproperty
        : property_namevalue ';'
        ;

channeldefinition
        : channelheader '{'
            opt_paramblock
          '}'
        ;

channelheader
        : CHANNEL NAME
           opt_inheritance
        ;

opt_inheritance
        : %empty
        | EXTENDS extendsname
        | LIKE likenames
        | EXTENDS extendsname LIKE likenames
        ;

extendsname
        : dottedname
        ;

likenames
        : likenames ',' likename
        | likename
        ;

likename
        : dottedname
        ;

channelinterfacedefinition
        : channelinterfaceheader '{'
            opt_paramblock
          '}'
        ;

channelinterfaceheader
        : CHANNELINTERFACE NAME
           opt_interfaceinheritance
        ;
```

```
opt_interfaceinheritance
        : EXTENDS extendsnames
        | %empty
        ;

extendsnames
        : extendsnames ',' extendsname
        | extendsname
        ;

simplemoduledefinition
        : simplemoduleheader '{'
            opt_paramblock
            opt_gateblock
          '}'
        ;

simplemoduleheader
        : SIMPLE NAME
          opt_inheritance
        ;

compoundmoduledefinition
        : compoundmoduleheader '{'
            opt_paramblock
            opt_gateblock
            opt_typeblock
            opt_submodblock
            opt_connblock
          '}'
        ;

compoundmoduleheader
        : MODULE NAME
          opt_inheritance
        ;

networkdefinition
        : networkheader '{'
            opt_paramblock
            opt_gateblock
            opt_typeblock
            opt_submodblock
            opt_connblock
          '}'
        ;

networkheader
        : NETWORK NAME
          opt_inheritance
        ;
```

```
moduleinterfacedefinition
        : moduleinterfaceheader '{'
            opt_paramblock
            opt_gateblock
          '}'
        ;

moduleinterfaceheader
        : MODULEINTERFACE NAME
           opt_interfaceinheritance
        ;

opt_paramblock
        : opt_params
        | PARAMETERS ':'
          opt_params
        ;

opt_params
        : params
        | %empty
        ;

params
        : params paramsitem
        | paramsitem
        ;

paramsitem
        : param
        | property
        ;

param
        : param_typenamevalue
        | pattern_value
        ;

param_typenamevalue
        : param_typename opt_inline_properties ';'
        | param_typename opt_inline_properties '=' paramvalue opt_inline_properties '
        ;

param_typename
        : opt_volatile paramtype NAME
        | NAME
        ;

pattern_value
        : pattern '=' paramvalue ';'
        ;
```

```
paramtype
        : DOUBLE
        | INT
        | STRING
        | BOOL
        | OBJECT
        | XML
        ;

opt_volatile
        : VOLATILE
        | %empty
        ;

paramvalue
        : expression
        | DEFAULT '(' expression ')'
        | DEFAULT
        | ASK
        ;

opt_inline_properties
        : inline_properties
        | %empty
        ;

inline_properties
        : inline_properties property_namevalue
        | property_namevalue
        ;

pattern
        : pattern2 '.' pattern_elem
        | pattern2 '.' TYPENAME
        ;

pattern2
        : pattern2 '.' pattern_elem
        | pattern_elem
        ;

pattern_elem
        : pattern_name
        | pattern_name '[' pattern_index ']'
        | pattern_name '[' '*' ']'
        | '**'
        ;

pattern_name
        : NAME
        | NAME '$' NAME
        | CHANNEL
```

```
        | '{' pattern_index '}'
        | '*'
        | pattern_name NAME
        | pattern_name '{' pattern_index '}'
        | pattern_name '*'
        ;

pattern_index
        : INTCONSTANT
        | INTCONSTANT '..' INTCONSTANT
        | '..' INTCONSTANT
        | INTCONSTANT '..'
        ;

property
        : property_namevalue ';'
        ;

property_namevalue
        : property_name
        | property_name '(' opt_property_keys ')'
        ;

property_name
        : '@' PROPNAME
        | '@' PROPNAME '[' PROPNAME ']'
        ;

opt_property_keys
        : property_keys
        ;

property_keys
        : property_keys ';' property_key
        | property_key
        ;

property_key
        : property_literal '=' property_values
        | property_values
        ;

property_values
        : property_values ',' property_value
        | property_value
        ;

property_value
        : property_literal
        | %empty
        ;
```

```
property_literal
        : property_literal CHAR
        | property_literal STRINGCONSTANT
        | CHAR
        | STRINGCONSTANT
        ;

opt_gateblock
        : gateblock
        | %empty
        ;

gateblock
        : GATES ':'
          opt_gates
        ;

opt_gates
        : gates
        | %empty
        ;

gates
        : gates gate
        | gate
        ;

gate
        : gate_typenamesize
          opt_inline_properties ';'
        ;

gate_typenamesize
        : gatetype NAME
        | gatetype NAME '[' ']'
        | gatetype NAME vector
        | NAME
        | NAME '[' ']'
        | NAME vector
        ;

gatetype
        : INPUT
        | OUTPUT
        | INOUT
        ;

opt_typeblock
        : typeblock
        | %empty
        ;
```

```
typeblock
        : TYPES ':'
           opt_localtypes
        ;

opt_localtypes
        : localtypes
        | %empty
        ;

localtypes
        : localtypes localtype
        | localtype
        ;

localtype
        : propertydecl
        | channeldefinition
        | channelinterfacedefinition
        | simplemoduledefinition
        | compoundmoduledefinition
        | networkdefinition
        | moduleinterfacedefinition
        | ';'
        ;

opt_submodblock
        : submodblock
        | %empty
        ;

submodblock
        : SUBMODULES ':'
          opt_submodules
        ;

opt_submodules
        : submodules
        | %empty
        ;

submodules
        : submodules submodule
        | submodule
        ;

submodule
        : submoduleheader ';'
        | submoduleheader '{'
          opt_paramblock
          opt_gateblock
          '}' opt_semicolon
```

```
        ;

submoduleheader
        : submodulename ':' dottedname opt_condition
        | submodulename ':' likeexpr LIKE dottedname opt_condition
        ;

submodulename
        : NAME
        |  NAME vector
        ;

likeexpr
        : '<' '>'
        | '<' expression '>'
        | '<' DEFAULT '(' expression ')' '>'
        ;

opt_condition
        : condition
        | %empty
        ;

opt_connblock
        : connblock
        | %empty
        ;

connblock
        : CONNECTIONS ALLOWUNCONNECTED ':'
          opt_connections
        | CONNECTIONS ':'
          opt_connections
        ;

opt_connections
        : connections
        | %empty
        ;

connections
        : connections connectionsitem
        | connectionsitem
        ;

connectionsitem
        : connectiongroup
        | connection opt_loops_and_conditions ';'
        ;

connectiongroup
        : opt_loops_and_conditions '{'
```

```
          connections '}' opt_semicolon
        ;

opt_loops_and_conditions
        : loops_and_conditions
        | %empty
        ;

loops_and_conditions
        : loops_and_conditions ',' loop_or_condition
        | loop_or_condition
        ;

loop_or_condition
        : loop
        | condition
        ;

loop
        : FOR NAME '=' expression '..' expression
        ;

connection
        : leftgatespec '-->' rightgatespec
        | leftgatespec '-->' channelspec '-->' rightgatespec
        | leftgatespec '<--' rightgatespec
        | leftgatespec '<--' channelspec '<--' rightgatespec
        | leftgatespec '<-->' rightgatespec
        | leftgatespec '<-->' channelspec '<-->' rightgatespec
        ;

leftgatespec
        : leftmod '.' leftgate
        | parentleftgate
        ;

leftmod
        : NAME vector
        | NAME
        ;

leftgate
        : NAME opt_subgate
        | NAME opt_subgate vector
        | NAME opt_subgate '++'
        ;

parentleftgate
        : NAME opt_subgate
        | NAME opt_subgate vector
        | NAME opt_subgate '++'
        ;
```

```
rightgatespec
        : rightmod '.' rightgate
        | parentrightgate
        ;

rightmod
        : NAME
        | NAME vector
        ;

rightgate
        : NAME opt_subgate
        | NAME opt_subgate vector
        | NAME opt_subgate '++'
        ;

parentrightgate
        : NAME opt_subgate
        | NAME opt_subgate vector
        | NAME opt_subgate '++'
        ;

opt_subgate
        : '$' NAME
        | %empty
        ;

channelspec
        : channelspec_header
        | channelspec_header '{'
            opt_paramblock
          '}'
        ;

channelspec_header
        : opt_channelname
        | opt_channelname dottedname
        | opt_channelname likeexpr LIKE dottedname
        ;

opt_channelname
        : %empty
        | NAME ':'
        ;

condition
        : IF expression
        ;

vector
        : '[' expression ']'
```

```
        ;

expression
        : expr
        ;

expr
        : simple_expr
        | functioncall
        | expr '.' functioncall
        | object
        | array
        | '(' expr ')'
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr '%' expr
        | expr '^' expr
        | '-' expr
                _
        | expr '==' expr
        | expr '!=' expr
        | expr '>' expr
        | expr '>=' expr
        | expr '<' expr
        | expr '<=' expr
        | expr '<=>' expr
        | expr '=~' expr
        | expr '&&' expr
        | expr '||' expr
        | expr '##' expr
        | '!' expr
                _
        | expr '&' expr
        | expr '|' expr
        | expr '#' expr
        | '~' expr
                _
        | expr '<<' expr
        | expr '>>' expr
        | expr '?' expr ':' expr
        ;

functioncall
        : funcname '(' opt_exprlist ')'
        ;

array
        : '[' ']'
        | '[' exprlist ']'
        | '[' exprlist ',' ']'
```

```
        ;

object
        : '{' opt_keyvaluelist '}'
        | NAME '{' opt_keyvaluelist '}'
        | NAME '::' NAME '{' opt_keyvaluelist '}'
        | NAME '::' NAME '::' NAME '{' opt_keyvaluelist '}'
        | NAME '::' NAME '::' NAME '::' NAME '{' opt_keyvaluelist '}'
        ;

opt_exprlist
        : exprlist
        | %empty
        ;

exprlist
        : exprlist ',' expr
        | expr
        ;

opt_keyvaluelist
        : keyvaluelist
        | keyvaluelist ','
        | %empty
        ;

keyvaluelist
        : keyvaluelist ',' keyvalue
        | keyvalue
        ;

keyvalue
        : key ':' expr
        ;

key
        : STRINGCONSTANT
        | NAME
        | INTCONSTANT
        | REALCONSTANT
        | quantity
        | '-' INTCONSTANT
        | '-' REALCONSTANT
        | '-' quantity
        | NAN
        | INF
        | '-' INF
        | TRUE
        | FALSE
        | NULL
        | NULLPTR
        ;
```

```
simple_expr
        : qname
        | operator
        | literal
        ;

funcname
        : NAME
        | BOOL
        | INT
        | DOUBLE
        | STRING
        | OBJECT
        | XML
        | XMLDOC
        ;

qname_elem
        : NAME
        | NAME '[' expr ']'
        | THIS
        | PARENT
        ;

qname
        : qname '.' qname_elem
        | qname_elem
        ;

operator
        : INDEX
        | TYPENAME
        | qname '.' INDEX
        | qname '.' TYPENAME
        | EXISTS '(' qname ')'
        | SIZEOF '(' qname ')'
        ;

literal
        : stringliteral
        | boolliteral
        | numliteral
        | otherliteral
        ;

stringliteral
        : STRINGCONSTANT
        ;

boolliteral
        : TRUE
```

```
        | FALSE
        ;

numliteral
        : INTCONSTANT
        | realconstant_ext
        | quantity
        ;

otherliteral
        : UNDEFINED
        | NULLPTR
        | NULL
        ;

quantity
        : quantity INTCONSTANT NAME
        | quantity realconstant_ext NAME
        | INTCONSTANT NAME
        | realconstant_ext NAME
        ;

realconstant_ext
        : REALCONSTANT
        | INF
        | NAN
        ;

opt_semicolon
        : ';'
        | %empty
        ;
```

Appendix C

NED XML Binding

This appendix shows the DTD for the XML binding of the NED language and message definitions.

```
<!ELEMENT ned-file (comment*, (package|import|property-decl|property|
                    simple-module|compound-module|module-interface|
                    channel|channel-interface)*)>
<!ATTLIST ned-file
     filename           CDATA     #REQUIRED
     version            CDATA     "2">

<!-- comments and whitespace; comments include '//' marks. Note that although
     nearly all elements may contain comment elements, there are places
     (e.g. within expressions) where they are ignored by the implementation.
     Default value is a space or a newline, depending on the context.
 -->
<!ELEMENT comment EMPTY>
<!ATTLIST comment
     locid              NMTOKEN   #REQUIRED
     content            CDATA     #IMPLIED>

<!ELEMENT package (comment*)>
<!ATTLIST package
     name               CDATA     #REQUIRED>

<!ELEMENT import (comment*)>
<!ATTLIST import
     import-spec        CDATA     #REQUIRED>

<!ELEMENT property-decl (comment*, property-key*, property*)>
<!ATTLIST property-decl
     name               CDATA     #REQUIRED
     is-array           (true|false) "false">

<!ELEMENT extends (comment*)>
<!ATTLIST extends
     name               CDATA     #REQUIRED>
```

```
<!ELEMENT interface-name (comment*)>
<!ATTLIST interface-name
     name               CDATA     #REQUIRED>

<!ELEMENT simple-module (comment*, extends?, interface-name*, parameters?, gates?)>
<!ATTLIST simple-module
     name               NMTOKEN   #REQUIRED>

<!ELEMENT module-interface (comment*, extends*, parameters?, gates?)>
<!ATTLIST module-interface
     name               NMTOKEN   #REQUIRED>

<!ELEMENT compound-module (comment*, extends?, interface-name*,
                           parameters?, gates?, types?, submodules?, connections?)>
<!ATTLIST compound-module
     name               NMTOKEN   #REQUIRED>

<!ELEMENT channel-interface (comment*, extends*, parameters?)>
<!ATTLIST channel-interface
     name                NMTOKEN   #REQUIRED>

<!ELEMENT channel (comment*, extends?, interface-name*, parameters?)>
<!ATTLIST channel
     name                NMTOKEN   #REQUIRED>

<!ELEMENT parameters (comment*, (property|param)*)>
<!ATTLIST parameters
    is-implicit         (true|false)  "false">

<!ELEMENT param (comment*, property*)>
<!ATTLIST param
     type               (double|int|string|bool|object|xml) #IMPLIED
     is-volatile        (true|false)  "false"
     name               CDATA     #REQUIRED
     value              CDATA     #IMPLIED
     is-pattern         (true|false)  "false"
     is-default         (true|false)  "false">

<!ELEMENT property (comment*, property-key*)>
<!ATTLIST property
     is-implicit        (true|false) "false"
     name               CDATA     #REQUIRED
     index              CDATA     #IMPLIED>

<!ELEMENT property-key (comment*, literal*)>
<!ATTLIST property-key
     name               CDATA     #IMPLIED>

<!ELEMENT gates (comment*, gate*)>

<!ELEMENT gate (comment*, property*)>
```

```
<!ATTLIST gate
     name               NMTOKEN   #REQUIRED
     type               (input|output|inout) #IMPLIED
     is-vector          (true|false) "false"
     vector-size        CDATA     #IMPLIED>

<!ELEMENT types (comment*, (channel|channel-interface|simple-module|
                            compound-module|module-interface)*)>

<!ELEMENT submodules (comment*, submodule*)>

<!ELEMENT submodule (comment*, condition?, parameters?, gates?)>
<!ATTLIST submodule
     name               NMTOKEN   #REQUIRED
     type               CDATA     #IMPLIED
     like-type          CDATA     #IMPLIED
     like-expr          CDATA     #IMPLIED
     is-default         (true|false) "false"
     vector-size        CDATA     #IMPLIED>

<!ELEMENT connections (comment*, (connection|connection-group)*)>
<!ATTLIST connections
     allow-unconnected (true|false) "false">

<!ELEMENT connection (comment*, parameters?, (loop|condition)*)>
<!ATTLIST connection
     src-module          NMTOKEN   #IMPLIED
     src-module-index    CDATA     #IMPLIED
     src-gate            NMTOKEN   #REQUIRED
     src-gate-plusplus  (true|false) "false"
     src-gate-index      CDATA     #IMPLIED
     src-gate-subg       (i|o)     #IMPLIED
     dest-module         NMTOKEN   #IMPLIED
     dest-module-index   CDATA     #IMPLIED
     dest-gate           NMTOKEN   #REQUIRED
     dest-gate-plusplus (true|false) "false"
     dest-gate-index     CDATA     #IMPLIED
     dest-gate-subg      (i|o)     #IMPLIED
     name                NMTOKEN   #IMPLIED
     type                CDATA     #IMPLIED
     like-type           CDATA     #IMPLIED
     like-expr           CDATA     #IMPLIED
     is-default         (true|false) "false"
     is-bidirectional   (true|false) "false"
     is-forward-arrow   (true|false) "true">

<!ELEMENT connection-group (comment*, (loop|condition)*, connection*)>

<!ELEMENT loop (comment*)>
<!ATTLIST loop
     param-name          NMTOKEN   #REQUIRED
     from-value          CDATA     #IMPLIED
```

```
     to-value            CDATA     #IMPLIED>

<!ELEMENT condition (comment*)>
<!ATTLIST condition
     condition           CDATA     #IMPLIED>

<!ELEMENT literal (comment*)>
<!-- Note: value is in fact REQUIRED, but empty attr value should
     also be accepted because that represents the "" string literal;
     "spec" is for properties, to store the null value and "-",
     the antivalue.
 -->
<!ATTLIST literal
     type  (double|quantity|int|bool|string|spec)  #REQUIRED
     text                CDATA     #IMPLIED
     value               CDATA     #IMPLIED>

<!--
  **  'unknown' is used internally to represent elements not in this NED DTD
  -->
<!ELEMENT unknown        ANY>
<!ATTLIST unknown
     element             CDATA     #REQUIRED>
```

Appendix D

NED Functions

The functions that can be used in NED expressions and ini files are the following. The question mark (as in "`rng?`") marks optional arguments.

D.1 Category "conversion":

bool : `bool bool(any x)`
Converts x to bool, and returns the result. For numeric values, 0 and nan become false and other values become true; for strings, "true" becomes true and everything else becomes false.

double : `quantity double(any x)`
Converts x to double, and returns the result. A boolean argument becomes 0 or 1; a string is interpreted as number; an object argument causes an error. Units are preserved.

int : `intquantity int(any x)`
Converts x to int, and returns the result. A boolean argument becomes 0 or 1; a double is converted using floor(); a string is interpreted as number; an object argument causes an error. Units are preserved.

string : `string string(any x)`
Converts x to string, and returns the result.

D.2 Category "i/o":

absFilePath : `string absFilePath(string filename)`
Converts filename to an absolute filesystem path. Absolute paths are returned unchanged; relative paths are understood as relative to the location of the ini or NED file where the call occurs (see baseDir()).

baseDir : `string baseDir()`
Returns the absolute filesystem path to the directory of the ini or NED file where the baseDir() call occurs.

parseCSV : `any parseCSV(string str)`
Parses the given string as a comma-separated CSV, and returns it as an array of arrays. Elements can be boolean (true/false), numeric (integer or double, with or without measurement unit), quoted string, or unquoted string. Items that cannot be parsed as any of the more specific types are interpreted as unquoted strings. See readCSV() for details of the the accepted CSV flavor.

parseExtendedCSV : `any parseExtendedCSV(string str)`
Parses the given string as a comma-separated CSV, and returns it as an array of arrays. Elements are parsed as NED expressions, and are evaluated in the caller's context. See readCSV() for details of the the accepted CSV flavor.

parseExtendedJSON : `any parseExtendedJSON(string str)`
Parses the given string as Extended JSON, and returns its contents. Extended JSON allows any value to be a valid NED expression (instead of just constants allowed by strict JSON), and some extensions to the object syntax. Actually, as the NED expression syntax includes JSON-like arrays and objects, "parsing" is done simply by evaluating the string as a NED expression in the caller's context.

parseJSON : `any parseJSON(string str)`
Parses the given string as JSON, and returns its contents. The syntax is more permissive than standard JSON: it additionally allows the special numeric values 'nan', 'inf' and '-inf', the use of measurement units, and object keys are also accepted without quotation marks if it doesn't interfere with parsing.

parseXML : `xml parseXML(string xmlstring, string xpath?)`
Parses the given XML string into a cXMLElement tree, and returns the root element. When called with two arguments, it returns the first element from the tree that matches the expression given in simplified XPath syntax.

readCSV : `any readCSV(string filename)`
Parses the content of the given text file as comma-separated CSV, and returns it as an array of arrays. Elements can be boolean ('true' or 'false'), numeric (integer or double, with or without measurement unit), quoted string, or unquoted string. Items that cannot be parsed as any of the more specific types are interpreted as unquoted strings. CSV parsing rules: separator is comma; blank (whitespace-only) lines are ignored; lines that contain hash mark '#' on column 1 are considered comments and are ignored; items are trimmed of leading and trailing whitespace before processing; no line continuation with backslash; quoted strings may be delimited with single or double quotes; quoted strings may contain C-like backslash escapes; no support for splitting strings over multiple lines; no special treatment for the first (possibly header) line.

readExtendedCSV : `any readExtendedCSV(string filename)`
Parses the content of the given text file as comma-separated CSV, and returns it as an array of arrays. Elements are parsed as NED expressions, and are evaluated in the caller's context. See readCSV() for details of the the accepted CSV flavor.

readExtendedJSON : `any readExtendedJSON(string filename)`
Parses the given text file as Extended JSON, and returns its contents. Extended JSON allows any value to be a valid NED expression (instead of just constants allowed by strict JSON), and some extensions to the object syntax. Actually, as the NED expression syntax includes JSON-like arrays and objects, "parsing" is done simply by evaluating the content as a NED expression in the caller's context.

readFile : `string readFile(string filename)`
Opens the specified text file, and returns its content as a string. If filename is a relative path, it is understood as relative to the location of the ini or NED file where the readFile() call occurs.

readJSON : `any readJSON(string filename)`
Parses the given text file as JSON, and returns its contents. The syntax is more permissive than standard JSON: it additionally allows the special numeric values 'nan', 'inf' and '-inf', the use of measurement units, and object keys are also accepted without quotation marks if it doesn't interfere with parsing.

readXML : `xml readXML(string filename, string xpath?)`
Parses the given XML file into a cXMLElement tree, and returns the root element. When called with two arguments, it returns the first element from the tree that matches the expression given in simplified XPath syntax.

resolveFile : `string resolveFile(string directory, string filename)`
Joins its arguments as file paths, except that when the second argument is an absolute filesystem path, it is returned unchanged. This is purely a string operation; neither directory nor filename needs to exist in the file system.

workingDir : `string workingDir()`
Returns the current working directory.

D.3 Category "math":

acos : `double acos(double)`
Trigonometric function; see standard C function of the same name

asin : `double asin(double)`
Trigonometric function; see standard C function of the same name

atan : `double atan(double)`
Trigonometric function; see standard C function of the same name

atan2 : `double atan2(double, double)`
Trigonometric function; see standard C function of the same name

ceil : `double ceil(double)`
Rounds up; see standard C function of the same name

cos : `double cos(double)`
Trigonometric function; see standard C function of the same name

exp : `double exp(double)`
Exponential; see standard C function of the same name

fabs : `quantity fabs(quantity x)`
Returns the absolute value of the quantity.

floor : `double floor(double)`
Rounds down; see standard C function of the same name

fmod : `quantity fmod(quantity x, quantity y)`
Returns the floating-point remainder of x/y; unit conversion takes place if needed.

hypot : `double hypot(double, double)`
Length of the hypotenuse; see standard C function of the same name

log : `double log(double)`
Natural logarithm; see standard C function of the same name

log10 : `double log10(double)`
Base-10 logarithm; see standard C function of the same name

max : `quantity max(quantity a, quantity b)`
Returns the greater one of the two quantities; unit conversion takes place if needed.

min : `quantity min(quantity a, quantity b)`
Returns the smaller one of the two quantities; unit conversion takes place if needed.

pow : `double pow(double, double)`
Power; see standard C function of the same name

sin : `double sin(double)`
Trigonometric function; see standard C function of the same name

sqrt : `double sqrt(double)`
Square root; see standard C function of the same name

tan : `double tan(double)`
Trigonometric function; see standard C function of the same name

D.4 Category "misc":

dup : `object dup(object obj)`
Clones the given object by calling its dup() C++ method.

eval : `any eval(string expr)`
Evaluates the NED expression in the call site's context.

firstAvailable : `string firstAvailable(...)`
Accepts any number of strings, interprets them as NED type names (qualified or unqualified), and returns the first one that exists and its C++ implementation class is also available. Throws an error if none of the types are available.

get : `any get(object arrayOrMap, any keyOrIndex)`
Obtain a value from a NED map or array. Examples: get([10,20,30], 1) returns 20; get(foo:10,bar:20, 'foo') returns 10. Note that get(x,i) may also be written as x.get(i), with the two being completely equivalent.

select : `any select(int index, ...)`
Returns the <index>th item from the rest of the argument list; numbering starts from 0.

simTime : `quantity simTime()`
Returns the current simulation time.

size : `int size(object arrayOrMap)`
Returns the length of an array, or the number of elements in a map.

D.5 Category "ned":

ancestorIndex : `int ancestorIndex(int numLevels)`
Returns the index of the ancestor module numLevels levels above the module or channel in context.

fullName : `string fullName()`
Returns the full name of the module or channel in context.

fullPath : `string fullPath()`
Returns the full path of the module or channel in context.

parentIndex : `int parentIndex()`
Returns the index of the parent module, which has to be part of module vector.

D.6 Category "random/continuous":

beta : `double beta(double alpha1, double alpha2, int rng?)`
Returns a random number from the Beta distribution

cauchy : `quantity cauchy(quantity a, quantity b, int rng?)`
Returns a random number from the Cauchy distribution

chi_square : `double chi_square(int k, int rng?)`
Returns a random number from the Chi-square distribution

erlang_k : `quantity erlang_k(int k, quantity mean, int rng?)`
Returns a random number from the Erlang distribution

exponential : `quantity exponential(quantity mean, int rng?)`
Returns a random number from the Exponential distribution

gamma_d : `quantity gamma_d(double alpha, quantity theta, int rng?)`
Returns a random number from the Gamma distribution

lognormal : `double lognormal(double m, double w, int rng?)`
Returns a random number from the Lognormal distribution

normal : `quantity normal(quantity mean, quantity stddev, int rng?)`
Returns a random number from the Normal distribution

pareto_shifted : `quantity pareto_shifted(double a, quantity b, quantity c, int rng?)`
Returns a random number from the Pareto-shifted distribution

student_t : `double student_t(int i, int rng?)`
Returns a random number from the Student-t distribution

triang : `quantity triang(quantity a, quantity b, quantity c, int rng?)`
Returns a random number from the Triangular distribution

truncnormal : `quantity truncnormal(quantity mean, quantity stddev, int rng?)`
Returns a random number from the truncated Normal distribution

uniform : `quantity uniform(quantity a, quantity b, int rng?)`
Returns a random number from the Uniform distribution

weibull : `quantity weibull(quantity a, quantity b, int rng?)`
Returns a random number from the Weibull distribution

D.7 Category "random/discrete":

bernoulli : `int bernoulli(double p, int rng?)`
Returns a random number from the Bernoulli distribution

binomial : `int binomial(int n, double p, int rng?)`
Returns a random number from the Binomial distribution

geometric : `int geometric(double p, int rng?)`
Returns a random number from the Geometric distribution

intuniform : `int intuniform(intquantity a, intquantity b, int rng?)`
Returns a random integer uniformly distributed over [a,b]

intuniformexcl : `int intuniformexcl(intquantity a, intquantity b, int rng?)`
Returns a random integer uniformly distributed over [a,b), that is, [a,b-1]

negbinomial : `int negbinomial(int n, double p, int rng?)`
Returns a random number from the Negbinomial distribution

poisson : `int poisson(double lambda, int rng?)`
Returns a random number from the Poisson distribution

D.8 Category "strings":

choose : `string choose(int index, string list)`
Interprets list as a space-separated list, and returns the item at the given index. Negative and out-of-bounds indices cause an error.

contains : `bool contains(string s, string substr)`
Returns true if string s contains substr as substring

endsWith : `bool endsWith(string s, string substr)`
Returns true if s ends with the substring substr.

expand : `string expand(string s)`
Expands $ variables ($configname, $runnumber, etc.) in the given string, and returns the result.

indexOf : `int indexOf(string s, string substr)`
Returns the position of the first occurrence of substring substr in s, or -1 if s does not contain substr.

length : `int length(string s)`
Returns the length of the string

replace : `string replace(string s, string substr, string repl, int startPos?)`
Replaces all occurrences of substr in s with the string repl. If startPos is given, search begins from position startPos in s.

replaceFirst : `string replaceFirst(string s, string substr, string repl, int startPos?)`
Replaces the first occurrence of substr in s with the string repl. If startPos is given, search begins from position startPos in s.

startsWith : `bool startsWith(string s, string substr)`
Returns true if s begins with the substring substr.

substring : `string substring(string s, int pos, int len?)`
Return the substring of s starting at the given position, either to the end of the string or maximum len characters

substringAfter : `string substringAfter(string s, string substr)`
Returns the substring of s after the first occurrence of substr, or the empty string if s does not contain substr.

substringAfterLast : `string substringAfterLast(string s, string substr)`
Returns the substring of s after the last occurrence of substr, or the empty string if s does not contain substr.

substringBefore : `string substringBefore(string s, string substr)`
Returns the substring of s before the first occurrence of substr, or the empty string if s does not contain substr.

substringBeforeLast : `string substringBeforeLast(string s, string substr)`
Returns the substring of s before the last occurrence of substr, or the empty string if s does not contain substr.

tail : `string tail(string s, int len)`
Returns the last len character of s, or the full s if it is shorter than len characters.

toLower : `string toLower(string s)`
Converts s to all lowercase, and returns the result.

toUpper : `string toUpper(string s)`
Converts s to all uppercase, and returns the result.

trim : `string trim(string s)`
Discards whitespace from the start and end of s, and returns the result.

D.9 Category "units":

convertUnit : `quantity convertUnit(quantity x, string unit)`
Converts x to the given unit.

dropUnit : `double dropUnit(quantity x)`
Removes the unit of measurement from quantity x.

replaceUnit : `quantity replaceUnit(quantity x, string unit)`
Replaces the unit of x with the given unit.

unitOf : `string unitOf(quantity x)`
Returns the unit of the given quantity.

D.10 Category "xml":

xml : `xml xml(string xmlstring, string xpath?)`
Parses the given XML string into a cXMLElement tree, and returns the root element. When called with two arguments, it returns the first element from the tree that matches the expression given in simplified XPath syntax. Note: This is an alias to the parseXML() function.

xmldoc : `xml xmldoc(string filename, string xpath?)`
Parses the given XML file into a cXMLElement tree, and returns the root element. When called with two arguments, it returns the first element from the tree that matches the expression given in simplified XPath syntax. Note: This is an alias to the readXML() function.

D.11 Category "units/conversion":

<unit_name> : `quantity <unit_name>(quantity x)`
All measurement unit names can be used as one-argument functions that convert from a compatible unit or a dimensionless number. Substitute underscore for any hyphen in the name, and '_per_' for any slash: milliampere-hour –> milliampere_hour(), meter/sec –> meter_per_sec().

d(), day(), h(), hour(), min(), minute(), s(), second(), ms(), millisecond(), us(), microsecond(), ns(), nanosecond(), ps(), picosecond(), fs(), femtosecond(), as(), attosecond(), bps(), bit_per_sec(), kbps(), kilobit_per_sec(), Mbps(), megabit_per_sec(), Gbps(), gigabit_per_sec(), Tbps(), terabit_per_sec(), B(), byte(), KiB(), kibibyte(), MiB(), mebibyte(), GiB(), gibibyte(), TiB(), tebibyte(), kB(), kilobyte(), MB(), megabyte(), GB(), gigabyte(), TB(), terabyte(), b(), bit(), Kib(), kibibit(), Mib(), mebibit(), Gib(), gibibit(), Tib(), tebibit(), kb(), kilobit(), Mb(), megabit(), Gb(), gigabit(), Tb(), terabit(), rad(), radian(), deg(), degree(), m(), meter(), cm(), centimeter(), mm(), millimeter(), um(), micrometer(), nm(), nanometer(), km(), kilometer(), W(), watt(), mW(), milliwatt(), uW(), microwatt(), nW(), nanowatt(), pW(), picowatt(), fW(), femtowatt(), kW(), kilowatt(), MW(), megawatt(), GW(), gigawatt(), Hz(), hertz(), kHz(), kilohertz(), MHz(), megahertz(), GHz(), gigahertz(), THz(), terahertz(), kg(), kilogram(), g(), gram(), K(), kelvin(), J(), joule(), kJ(), kilojoule(), MJ(), megajoule(), Ws(), watt_second(), Wh(), watt_hour(), kWh(), kilowatt_hour(), MWh(), megawatt_hour(), V(), volt(), kV(), kilovolt(), mV(), millivolt(), A(), ampere(), mA(), milliampere(), uA(), microampere(), Ohm(), ohm(), mOhm(), milliohm(), kOhm(), kiloohm(), MOhm(), megaohm(), mps(), meter_per_sec(), kmps(), kilometer_per_sec(), kmph(), kilometer_per_hour(), C(), coulomb(), As(), ampere_second(), mAs(), milliampere_second(), Ah(), ampere_hour(), mAh(), milliampere_hour(), ratio(), ratio(), pct(), percent(), dBW(), decibel_watt(), dBm(), decibel_milliwatt(), dBmW(), decibel_milliwatt(), dBV(), decibel_volt(), dBmV(), decibel_millivolt(), dBA(), decibel_ampere(), dBmA(), decibel_milliampere(), dB(), decibel(), etc.

Appendix E

Message Definitions Grammar

This appendix contains the grammar for the message definitions language.

In the language, space, horizontal tab and new line characters count as delimiters, so one or more of them is required between two elements of the description which would otherwise be unseparable.

'//' (two slashes) may be used to write comments that last to the end of the line.

The language is fully case sensitive.

Notation:

- rule syntax is that of *bison*
- uppercase words are terminals, lowercase words are nonterminals
- `NAME`, `CHARCONSTANT`, `STRINGCONSTANT`, `INTCONSTANT`, `REALCONSTANT` represent identifier names and string, character, integer and real number literals (defined as in the C language)
- other terminals represent keywords in all lowercase

Nonterminals ending in `_old` are present so that message files from OMNEST (3.x) can be parsed.

```
msgfile
        : definitions
        ;

definitions
        : definitions definition
        | %empty
        ;

definition
        : namespace_decl
        | fileproperty
        | cplusplus
        | import
```

```
        | struct_decl
        | class_decl
        | message_decl
        | packet_decl
        | enum_decl
        | enum
        | message
        | packet
        | class
        | struct
        ;

namespace_decl
        : NAMESPACE qname ';'
        | NAMESPACE ';'

qname
        : '::' qname1
        | qname1
        ;

qname1
        : qname1 '::' NAME
        | NAME
        ;

fileproperty
        : property_namevalue ';'
        ;

cplusplus
        : CPLUSPLUS '{{' ... '}}' opt_semicolon
        | CPLUSPLUS '(' targetspec ')' '{{' ... '}}' opt_semicolon
        ;
        ;

targetspec
        : targetspec targetitem
        | targetitem
        ;

targetitem
        : NAME | '::' | INTCONSTANT | ':' | '.' | ',' | '~' | '=' | '&'
        ;

import
        : IMPORT importspec ';'
        ;

importspec
        : importspec '.' importname
        | importname
```

```
        ;

importname
        : NAME
        | MESSAGE | PACKET | CLASS | STRUCT | ENUM | ABSTRACT
        ;

struct_decl
        : STRUCT qname ';'
        ;

class_decl
        : CLASS qname ';'
        | CLASS NONCOBJECT qname ';'
        | CLASS qname EXTENDS qname ';'
        ;

message_decl
        : MESSAGE qname ';'
        ;

packet_decl
        : PACKET qname ';'
        ;

enum_decl
        : ENUM qname ';'
        ;

enum
        : ENUM qname '{'
          opt_enumfields '}' opt_semicolon
        ;

opt_enumfields
        : enumfields
        | %empty
        ;

enumfields
        : enumfields enumfield
        | enumfield
        ;

enumfield
        : NAME ';'
        | NAME '=' enumvalue ';'
        ;

message
        : message_header body
        ;
```

```
packet
        : packet_header body
        ;

class
        : class_header body
        ;

struct
        : struct_header body
        ;

message_header
        : MESSAGE qname '{'
        | MESSAGE qname EXTENDS qname '{'
        ;

packet_header
        : PACKET qname '{'
        | PACKET qname EXTENDS qname '{'
        ;

class_header
        : CLASS qname '{'
        | CLASS qname EXTENDS qname '{'
        ;

struct_header
        : STRUCT qname '{'
        | STRUCT qname EXTENDS qname '{'
        ;

body
        : opt_fields_and_properties
          '}' opt_semicolon
        ;

opt_fields_and_properties
        : fields_and_properties
        | %empty
        ;

fields_and_properties
        : fields_and_properties field
        | fields_and_properties property
        | field
        | property
        ;

field
        :  fieldtypename opt_fieldvector opt_inline_properties ';'
```

```
        |  fieldtypename opt_fieldvector opt_inline_properties '=' fieldvalue opt_inl
        ;

fieldtypename
        : fieldmodifiers fielddatatype NAME
        | fieldmodifiers NAME
        ;

fieldmodifiers
        : ABSTRACT
        | %empty
        ;

fielddatatype
        : fieldsimpledatatype
        | fieldsimpledatatype '*'
        | CONST fieldsimpledatatype
        | CONST fieldsimpledatatype '*'
        ;

fieldsimpledatatype
        : qname
        | CHAR
        | SHORT
        | INT
        | LONG
        | UNSIGNED CHAR
        | UNSIGNED SHORT
        | UNSIGNED INT
        | UNSIGNED LONG
        | DOUBLE
        | STRING
        | BOOL
        ;

opt_fieldvector
        : '[' INTCONSTANT ']'
        | '[' qname ']'
        | '[' ']'
        | %empty
        ;

fieldvalue
        : fieldvalue fieldvalueitem
        | fieldvalueitem
        ;

fieldvalueitem
        : STRINGCONSTANT
        | CHARCONSTANT
        | INTCONSTANT
        | REALCONSTANT
```

```
        | TRUE
        | FALSE
        | NAME
        | '::'
        | '?' | ':' | '&&' | '||' | '##' | '==' | '!=' | '>' | '>=' | '<' | '<='
        | '&' | '|' | '#' | '<<' | '>>'
        | '+' | '-' | '*' | '/' | '%' | '^' | UMIN | '!' | '~'
        | '.' | ',' | '(' | ')' | '[' | ']'
        ;

enumvalue
        : INTCONSTANT
        | '-' INTCONSTANT
        | NAME
        ;

opt_inline_properties
        : inline_properties
        | %empty
        ;

inline_properties
        : inline_properties property_namevalue
        | property_namevalue
        ;

property
        : property_namevalue ';'
        ;

property_namevalue
        : property_name
        | property_name '(' opt_property_keys ')'
        | ENUM '(' NAME ')'
        ;

property_name
        : '@' PROPNAME
        | '@' PROPNAME '[' PROPNAME ']'
        ;

opt_property_keys
        : property_keys
        ;

property_keys
        : property_keys ';' property_key
        | property_key
        ;

property_key
        : property_literal '=' property_values
```

```
        | property_values
        ;

property_values
        : property_values ',' property_value
        | property_value
        ;

property_value
        : property_literal
        | %empty
        ;

property_literal
        : property_literal CHAR
        | property_literal STRINGCONSTANT
        | CHAR
        | STRINGCONSTANT
        ;

opt_semicolon
        : ';'
        | %empty
        ;
```

Appendix F

Message Class/Field Properties

This appendix lists the properties that can be used to customize C++ code generated from message descriptions.

@abstract *(type: bool, use: field, class)*
If true: For a class, it indicates that it is an abstract class in the C++ sense (one which cannot be instantiated). For a field, it requests that the accessor methods for the field be made pure virtual and no data member be generated; it also makes the class that contains the field abstract (unless the class has @customize whereas it is assumed that the custom code implements the pure virtual member functions).

@allowReplace *(type: bool, use: field)*
Specifies whether the setter method of an owned pointer field is allowed to delete the previously set object.

@appender *(type: string, use: field)*
Name of the appender method. (This method appends an element to a dynamic array.) When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@argType *(type: string, use: field, class)*
Field setter argument C++ base type. This type may be decorated with 'const' and '*'/'&' to produce the actual argument type. When specified on a class, it determines the default for fields of that type.

@beforeChange *(type: string, use: class)*
Method to be called before mutator code (in setters, non-const getters, operator=, etc.).

@byValue *(type: bool, use: field, class)*
If true: Causes the value to be passed by value (instead of by reference) in setters/getters. When specified on a class, it determines the default for fields of that type.

@castFunction *(type: bool, use: class)*
If false: Do not specialize the fromAnyPtr<T>(any_ptr) function for this class. Useful for preventing compile errors if the function already exists, e.g. in hand-written form, or generated for another type (think aliased typedefs).

@clone *(type: string, use: field, class)*
For owned pointer fields: Code to duplicate (one array element of) the field value. When specified on a class, it determines the default for fields of that type.

@cppType *(type: string, use: field, class)*
C++ datatype. Provides a common default for @datamemberType, @argType and @returnType. When specified on a class, it determines the default for fields of that type.

@custom *(type: bool, use: field)*
If true: Do not generate any data or code for the field, only add it to the descriptor. Indicates that the field's implementation will be added to the class via targeted cplusplus blocks.

@customImpl *(type: bool, use: field)*
If true: Do not generate implementations for the field's accessor methods, to allow custom implementations to be supplied by the user via cplusplus blocks or in separate .cc files.

@customize *(type: bool, use: class)*
If true: Customize the class via inheritance. Generates base class <name>_Base.

@datamemberType *(type: string, use: field, class)*
Data member C++ base data type. This type is decorated with '*' if the field is a pointer. When specified on a class, it determines the default for fields of that type.

@defaultValue *(type: string, use: class)*
Default value for fields of this type.

@descriptor *(type: string, use: class)*
A 'true'/'false' value specifies whether to generate descriptor class; special value 'readonly' requests generating a read-only descriptor (but specifying @editable/@replaceable/@resizable on individual fields overrides that).

@editable *(type: bool, use: field, class)*
Affects descriptor class only. If true: Value of the field (or value of fields that are instances of this type) can be set via the class descriptor's setFieldValueFromString() and setFieldValue() methods.

@enum *(type: string, use: field)*
For integer fields: Values are from the given enum.

@eraser *(type: string, use: field)*
Name of the eraser method. (This method erases an element from a dynamic array. Indices above the specified one are shifted down.) When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@eventlog *(type: string, use: field)*
When @eventlog(skip) is given, eventlog recording will skip this field when serializing objects

@existingClass *(type: bool, use: class)*
If true: This is a type is already defined in C++, i.e. it does not need to be generated.

@fieldNameSuffix *(type: string, use: class)*
Suffix to append to the names of data members.

@fromString *(type: string, use: field, class)*
Affects descriptor class only. Method name, code fragment or funcall template to convert string to field value. When specified on a class, it determines the default for fields of that type.

@fromValue *(type: string, use: field, class)*
Affects descriptor class only. Method name, code fragment or funcall template to convert cValue to field value. When specified on a class, it determines the default for fields of that type.

@getter *(type: string, use: field)*
Name of the (const) getter method. When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@getterConversion *(type: string, use: field, class)*
Method name, code fragment or funcall template to convert field data type to the return type in getters. When specified on a class, it determines the default for fields of that type.

@getterForUpdate *(type: string, use: field)*
Name of the non-const getter method. When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@group *(type: string, use: field)*
Used for grouping of fields in Qtenv inspectors

@hint *(type: string, use: field)*
Short description of the field, displayed in Qtenv inspectors as tooltip

@icon *(type: string, use: class)*
Icon for objects of this class in Qtenv inspectors

@implements *(type: stringlist, use: class)*
Names of additional base classes.

@inserter *(type: string, use: field)*
Name of the inserter method. (This method inserts an element into a dynamic array.) When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@label *(type: string, use: field)*
When specified, this string will be displayed as field name in Qtenv inspectors

@nopack *(type: bool, use: field)*
If true: Ignore this field in parsimPack/parsimUnpack methods.

@omitGetVerb *(type: bool, use: class)*
If true: Drop the 'get' verb from the names of getter methods.

@opaque *(type: bool, use: field, class)*
Affects descriptor class only. If true: Treat the field as atomic (non-compound) type, i.e. having no descriptor class. When specified on a class, it determines the default for fields of that type.

@overrideGetter *(type: bool, use: field)*
If true: Add the 'override' keyword to the declaration of the getter method.

@overrideSetter *(type: bool, use: field)*
If true: Add the 'override' keyword to the declaration of the setter method.

@owned *(type: bool, use: field)*
For pointers and pointer arrays: Whether allocated memory is owned by the object (needs to be duplicated in dup(), and deleted in destructor). If field type is also a cOwnedObject, take()/drop() calls are also generated.

@packetData *(type: string, use: class, field)*
Denotes packet data in frameworks such as INET; used in Qtenv inspectors

@polymorphic *(type: bool, use: class)*
Specifies whether this type is polymorphic, i.e. has any virtual member function.

@primitive *(type: bool, use: field, class)*
Shortcut for @opaque @byValue @editable @subclassable(false) @supportsPtr(false).

@property *(type: any, use: file)*
Property for declaring properties.

@readonly *(type: bool, use: field)*
Affects descriptor class only. Equivalent to @editable(false) @replaceable(false) @resizable(false).

@remover *(type: string, use: field)*
Name of the remover method. (This method is generated for owned pointer fields. It releases the ownership of the stored object, sets the field to nullptr, then returns the object.) When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@replaceable *(type: bool, use: field)*
Affects descriptor class only. If true: Field is a pointer whose value can be set via the class descriptor's setFieldStructValuePointer() and setFieldValue() methods.

@resizable *(type: bool, use: field)*
Affects descriptor class only. If true: Field is a variable-size array whose size can be set via the class descriptor's setFieldArraySize() method.

@returnType *(type: string, use: field, class)*
Field getter C++ base return type. This type may be decorated with 'const' and '*'/'&' to produce the actual return type. When specified on a class, it determines the default for fields of that type.

@setter *(type: string, use: field)*
Name of the setter method. When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@sizeGetter *(type: string, use: field)*
Name of the method that returns the array size. When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@sizeSetter *(type: string, use: field)*
Name of the method that sets size of dynamic array. When generating a descriptor for an existing class (see @existingClass), a code fragment or funcall template for the equivalent functionality is also accepted.

@sizeType *(type: string, use: field)*
C++ type to use for array sizes and indices.

@str *(type: string, use: class)*
Expression to be returned from the generated str() method.

@subclassable *(type: bool, use: class)*
Specifies whether this type can be subclassed (e.g. C++ primitive types and final classes cannot).

@supportsPtr *(type: bool, use: field, class)*
Specifies whether this type supports creating a pointer (or pointer array) from it.

@toString *(type: string, use: field, class)*
Affects descriptor class only. Method name, code fragment or funcall template to convert field value to string. When specified on a class, it determines the default for fields of that type.

@toValue *(type: string, use: field, class)*
Affects descriptor class only. Method name, code fragment or funcall template to convert field value to cValue. When specified on a class, it determines the default for fields of that type.

Appendix G

Display String Tags

G.1 Module and Connection Display String Tags

Supported module and connection display string tags are listed in the following table.

Tag[argument index] - name	Description
p[0] - x	X position of the center of the icon/shape; defaults to automatic graph layouting
p[1] - y	Y position of the center of the icon/shape; defaults to automatic graph layouting
p[2] - arrangement	Arrangement of submodule vectors. Values: row (r), column (c), matrix (m), ring (ri), exact (x)
p[3] - arr. par1	Depends on arrangement: matrix => ncols, ring => rx, exact => dx, row => dx, column => dy
p[4] - arr. par2	Depends on arrangement: matrix => dx, ring => ry, exact => dy
p[5] - arr. par3	Depends on arrangement: matrix => dy
g[5] - layout group	Allows unrelated modules to be arranged in a row, column, matrix, etc
b[0] - width	Width of object. Default: 40
b[1] - height	Height of object. Default: 24
b[2] - shape	Shape of object. Values: rectangle (rect), oval (oval). Default: rect
b[3] - fill color	Fill color of the object (color name, #RRGGBB or @HHSSBB). Default: #8080ff
b[4] - border color	Border color of the object (color name, #RRGGBB or @HHSSBB). Default: black
b[5] - border width	Border width of the object. Default: 2
i[0] - icon	An icon representing the object
i[1] - icon tint	A color for tinting the icon (color name, #RRGGBB or @HHSSBB)
i[2] - icon tint	Amount of tinting in percent. Default: 30
is[0] - icon size	The size of the image. Values: very small (vs), small (s), normal (n), large (l), very large (vl)

i2[0] - overlay icon	An icon added to the upper right corner of the original image
i2[1] - overlay icon tint	A color for tinting the overlay icon (color name, #RRGGBB or @HHSSBB)
i2[2] - overlay icon tint	Amount of tinting in percent. Default: 30
r[0] - range	Radius of the range indicator
r[1] - range fill color	Fill color of the range indicator (color name, #RRGGBB or @HHSSBB)
r[2] - range border color	Border color of the range indicator (color name, #RRGGBB or @HHSSBB). Default: black
r[3] - range border width	Border width of the range indicator. Default: 1
q[0] - queue object	Displays the length of the named queue object
t[0] - text	Additional text to display
t[1] - text position	Position of the text. Values: left (l), right (r), top (t). Default: t
t[2] - text color	Color of the displayed text (color name, #RRGGBB or @HHSSBB). Default: blue
tt[0] - tooltip	Tooltip to be displayed over the object
bgb[0] - bg width	Width of the module background rectangle
bgb[1] - bg height	Height of the module background rectangle
bgb[2] - bg fill color	Background fill color (color name, #RRGGBB or @HHSSBB). Default: grey82
bgb[3] - bg border color	Border color of the module background rectangle (color name, #RRGGBB or @HHSSBB). Default: black
bgb[4] - bg border width	Border width of the module background rectangle. Default: 2
bgtt[0] - bg tooltip	Tooltip to be displayed over the module's background
bgi[0] - bg image	An image to be displayed as a module background
bgi[1] - bg image mode	How to arrange the module's background image. Values: fix (f), tile (t), stretch (s), center (c). Default: fixed
bgg[0] - grid distance	Distance between two major gridlines, in units
bgg[1] - grid subdivision	Minor gridlines per major gridlines. Default: 1
bgg[2] - grid color	Color of the grid lines (color name, #RRGGBB or @HHSSBB). Default: grey
bgu[0] - distance unit	Name of distance unit. Default: m
m[0] - routing constraint	Connection routing constraint. Values: auto (a), south (s), north (n), east (e), west (w), manual (m)
m[1] - src anchor x	When m[0] is 'm', this is the x coordinate of one point of the connection line, in integer percentages of the source rectangle
m[2] - src anchor y	When m[0] is 'm', this is the y coordinate of one point of the connection line, in integer percentages of the source rectangle

m[3] - dest anchor x	When m[0] is 'm', this is the x coordinate of another point of the connection line, in integer percentages of the destination rectangle
m[4] - dest anchor y	When m[0] is 'm', this is the y coordinate of another point of the connection line, in integer percentages of the destination rectangle
ls[0] - line color	Connection color (color name, #RRGGBB or @HHSSBB). Default: black
ls[1] - line width	Connection line width. Default: 1
ls[2] - line style	Connection line style. Values: solid (s), dotted (d), dashed (da). Default: solid

G.2 Message Display String Tags

To customize the appearance of messages in the graphical runtime environment, override the `getDisplayString()` method of `cMessage` or `cPacket` to return a display string.

Tag	Meaning
b=*width*,*height*,**oval**	Ellipse with the given *height* and *width*. Defaults: *width*=10, *height*=10
b=*width*,*height*,**rect**	Rectangle with the given *height* and *width*. Defaults: *width*=10, *height*=10
o=*fillcolor*,*outlinecolor*,*borderwidth*	Specifies options for the rectangle or oval. Colors can be given in HTML format (#rrggbb), in HSB format (@hhssbb), or as a valid SVG color name. Defaults: *fillcolor*=red, *outlinecolor*=black, *borderwidth*=1
i=*iconname*,*color*,*percentage*	Use the named icon. It can be colorized, and percentage specifies the amount of colorization. If color name is `"kind"`, a message kind dependent colors is used (like default behaviour). Defaults: *iconname*: no default – if no icon name is present, a small red solid circle will be used; *color*: no coloring; *percentage*: 30%
tt=*tooltip-text*	Displays the given text in a tooltip when the user moves the mouse over the message icon.

Appendix H

Figure Definitions

This appendix provides a reference to defining figures in NED files.

H.1 Built-in Figure Types

The following table lists the figure types supported by OMNEST.

@figure type	C++ class
line	`cLineFigure`
arc	`cArcFigure`
polyline	`cPolylineFigure`
rectangle	`cRectangleFigure`
oval	`cOvalFigure`
ring	`cRingFigure`
pieslice	`cPieSliceFigure`
polygon	`cPolygonFigure`
path	`cPathFigure`
text	`cTextFigure`
label	`cLabelFigure`
image	`cImageFigure`
icon	`cIconFigure`
pixmap	`cPixmapFigure`
group	`cGroupFigure`

Additional figure types can be defined with the `custom:` *<type>* syntax; see *FigureType* below.

H.2 Attribute Types

This section lists what attribute types exist and their value syntaxes.

bool :
`true` or `false`.

int :
An integer.

double :
A real number.

double01 :
A real number in the [0,1] interval.

degrees :
A real number that will be interpreted as degrees.

string :
A string. It only needs to be enclosed in quotes if it contains comma, semicolon, unmatched close parenthesis or other character that prevents it from being parsed properly as a property value.

Anchor :
`c`, `center`, `n`, `e`, `s`, `w`, `nw`, `ne`, `se`, `sw`, `start`, `middle`, or `end`. The last three are only valid for text figures.

Arrowhead :
`none`, `simple`, `triangle`, or `barbed`.

CapStyle :
`butt`, `square`, or `round`.

Color :
A color in HTML format (*#rrggbb*), a color in HSB format (*@hhssbb*), or a valid SVG color name.

Dimensions : *width, height*
Size given as width and height.

FigureType :
One of the built-in figure types (e.g. `line` or `arc`, see H.1), or a figure type registered with `Register_Figure()`.

FillRule :
`evenodd` or `nonzero`.

Font : *typeface, size, style*
All three items are optional. *size* is the font size in points. *style* is space-sparated list of zero or more of the following words: `normal`, `bold`, `italic`, `underline`.

ImageName :
The name of an image.

Interpolation :
`none`, `fast`, or `best`.

JoinStyle :
`bevel`, `miter`, or `round`.

LineStyle :
`solid`, `dotted`, or `dashed`.

Point : *x, y*
A point with (x, y) coordinates.

Point2 : *x1, y1, x2, y2*
Two points: $(x1, y1)$ and $(x2, y2)$.

PointList : *x1, y1, x2, y2, x3, y3...*
A list of the $(x1, y1)$, $(x2, y2)$, $(x3, y3)$, etc. points.

Rectangle : *x, y, width, height*
A rectangle given with its top-left corner and dimensions.

TagList : *tag1, tag2, tag3...*
A list of string tags.

Tint : *Color, double01*
Specifies tint color and the amount of tinting for images.

Transform :
One or more transform steps. A step is one of:
`translate(`x`, `y`)`,
`rotate(`deg`)`,
`rotate(`deg`, `$centerx$`, `$centery$`)`,
`scale(`s`)`, `scale(`sx`, `sy`)`,
`scale(`s`, `$centerx$`, `$centery$`)`,
`scale(`sx`, `sy`, `$centerx$`, `$centery$`)`,
`skewx(`$coeff$`)`,
`skewx(`$coeff$`, `$centery$`)`,
`skewy(`$coeff$`)`,
`skewy(`$coeff$`, `$centerx$`)`,
`matrix(`a`, `b`, `c`, `d`, `$t1$`, `$t2$`)`

H.3 Figure Attributes

This section lists what attributes are accepted by individual figure types. Types enclosed in parentheses are abstract types which cannot be used directly; their sole purpose is to provide a base for more specialized types.

(figure) :
`type`=<*FigureType*>; `visible`=<*bool*>; `tags`=<*TagList*>; `childZ`=<*int*>;
`transform`=<*Transform*>;

(abstractLine) : figure
`lineColor`=<*Color*>; `lineStyle`=<*LineStyle*>; `lineWidth`=<*double*>;
`lineOpacity`=<*double*>; `capStyle`=<*CapStyle*>; `startArrowhead`=<*Arrowhead*>;
`endArrowhead`=<*Arrowhead*>; `zoomLineWidth`=<*bool*>;

line : abstractLine
`points`=<*Point2*>

arc : abstractLine
`bounds`=<*Rectangle*> `pos`=<*Point*>; `size`=<*Dimensions*>; `anchor`=<*Anchor*>;
`startAngle`=<*degrees*>; `endAngle`=<*degrees*>

polyline : abstractLine
`points`=<*PointList*>; `smooth`=<*bool*>; `joinstyle`=<*JoinStyle*>

(abstractShape) : figure
`lineColor`=<*Color*>; `fillColor`=<*Color*>; `lineStyle`=<*LineStyle*>; `lineWidth`=<*double*>; `lineOpacity`=<*double01*>; `fillOpacity`=<*double01*>; `zoomLineWidth`=<*bool*>

rectangle : abstractShape
`bounds`=<*Rectangle*> `pos`=<*Point*>; `size`=<*Dimensions*>; `anchor`=<*Anchor*>; `cornerRadius`=<*double*>|<*Dimensions*>

oval : abstractShape
`bounds`=<*Rectangle*> `pos`=<*Point*>; `size`=<*Dimensions*>; `anchor`=<*Anchor*>

ring : abstractShape
`bounds`=<*Rectangle*> `pos`=<*Point*>; `size`=<*Dimensions*>; `anchor`=<*Anchor*>; `innerSize`=<*Dimensions*>

pieslice : abstractShape
`bounds`=<*Rectangle*> `pos`=<*Point*>; `size`=<*Dimensions*>; `anchor`=<*Anchor*>; `startAngle`=<*degrees*>; `endAngle`=<*degrees*>

polygon : abstractShape
`points`=<*PointList*>; `smooth`=<*bool*>; `joinStyle`=<*JoinStyle*>; `fillRule`=<*FillRule*>

path : abstractShape
`path`=<*string*>; `offset`=<*Point*>; `joinStyle`=<*JoinStyle*>; `capStyle`=<*CapStyle*>; `fillRule`=<*FillRule*>

(abstractText) : figure
`pos`=<*Point*>; `anchor`=<*Anchor*> `text`=<*string*>; `font`=<*Font*>; `opacity`=<*double01*>; `color`=<*Color*>;

label : abstractText
`angle`=<*degrees*>;

text : abstractText

(abstractImage) : figure
`bounds`=<*Rectangle*> `pos`=<*Point*>; `size`=<*Dimensions*>; `anchor`=<*Anchor*>; `interpolation`=<*Interpolation*>; `opacity`=<*double01*>; `tint`=<*Tint*>

image : abstractImage
`image`=<*ImageName*>

icon : abstractImage
`image`=<*ImageName*>

pixmap : abstractImage
`resolution`=<*Dimensions*>

Appendix I

Configuration Options

I.1 Configuration Options

This section lists all configuration options that are available in ini files. A similar list can be obtained from any simulation executable by running it with the `-h configdetails` option.

allow-object-stealing-on-deletion = <*bool*>, default: `false`
Per-simulation-run setting.
Setting it to true disables the "Context component is deleting an object it doesn't own" error message. This option exists primarily for backward compatibility with pre-6.0 versions that were more permissive during object deletion.

****.bin-recording** = <*bool*>, default: `true`
Per-object setting for scalar results.
Whether the bins of the matching histogram object should be recorded, provided that recording of the histogram object itself is enabled (`**.<scalar-name>.scalar-recording=true`).
Usage: `<module-full-path>.<scalar-name>.bin-recording=true/false`. To control histogram recording from a `@statistic`, use `<statistic-name>:histogram` for `<scalar-name>`.
Example: `**.ping.roundTripTime:histogram.bin-recording=false`

check-signals = <*bool*>, default: `true`
Per-simulation-run setting.
Controls whether the simulation kernel will validate signals emitted by modules and channels against signal declarations (`@signal` properties) in NED files. The default setting depends on the build type: `true` in DEBUG, and `false` in RELEASE mode.

cmdenv-autoflush = <*bool*>, default: `false`
Per-simulation-run setting.
Call `fflush(stdout)` after each event banner or status update; affects both express and normal mode. Turning on autoflush may have a performance penalty, but it can be useful with printf-style debugging for tracking down program crashes.

cmdenv-config-name = <*string*>
Global setting (applies to all simulation runs).
Specifies the name of the configuration to be run (for a value `Foo`, section `[Config Foo]`

will be used from the ini file). See also `cmdenv-runs-to-execute`. The `-c` command line option overrides this setting.

cmdenv-event-banner-details = <*bool*>, default: `false`
Per-simulation-run setting.
When `cmdenv-express-mode=false`: print extra information after event banners.

cmdenv-event-banners = <*bool*>, default: `true`
Per-simulation-run setting.
When `cmdenv-express-mode=false`: turns printing event banners on/off.

cmdenv-express-mode = <*bool*>, default: `true`
Per-simulation-run setting.
Selects normal (debug/trace) or express mode.

cmdenv-extra-stack = <*double*>, unit=`B`, default: `8KiB`
Global setting (applies to all simulation runs).
Specifies the extra amount of stack that is reserved for each `activity()` simple module when the simulation is run under Cmdenv.

cmdenv-fake-gui = <*bool*>, default: `false`
Per-simulation-run setting.
Causes Cmdenv to lie to simulations that is a GUI (isGui()=true), and to periodically invoke refreshDisplay() during simulation execution.

cmdenv-fake-gui-after-event-probability = <*double*>, default: `1`
Per-simulation-run setting.
When `cmdenv-fake-gui=true`: The probability with which `refreshDisplay()` is called after each event.

cmdenv-fake-gui-before-event-probability = <*double*>, default: `1`
Per-simulation-run setting.
When `cmdenv-fake-gui=true`: The probability with which `refreshDisplay()` is called before each event.

cmdenv-fake-gui-on-hold-numsteps = <*custom*>, default: `3`
Per-simulation-run setting.
When `cmdenv-fake-gui=true`: The number of times `refreshDisplay()` is called during a "hold" period (animation during which simulation time does not advance), provided a trial with `cmdenv-fake-gui-on-hold-probability` yielded success. This an expression which will be evaluated each time, so it can be random. Zero is also a valid value.

cmdenv-fake-gui-on-hold-probability = <*double*>, default: `0.5`
Per-simulation-run setting.
When `cmdenv-fake-gui=true`: The probability with which `refreshDisplay()` is called (possibly multiple times, see `cmdenv-fake-gui-on-hold-numsteps`) during a "hold" period (animation during which simulation time does not advance).

cmdenv-fake-gui-on-simtime-numsteps = <*custom*>, default: `3`
Per-simulation-run setting.
When `cmdenv-fake-gui=true`: The number of times `refreshDisplay()` is called when simulation time advances from one simulation event to the next, provided a trial with `cmdenv-fake-gui-on-simtime-probability` yielded success. This an expression which will be evaluated each time, so it can be random. Zero is also a valid value.

cmdenv-fake-gui-on-simtime-probability = <*double*>, default: `0.1`
Per-simulation-run setting.
When `cmdenv-fake-gui=true`: The probability with which `refreshDisplay()` is called (possibly multiple times, see `cmdenv-fake-gui-on-simtime-numsteps`) when simulation time advances from one simulation event to the next.

cmdenv-fake-gui-seed = <*int*>, default: `1`
Per-simulation-run setting.
When `cmdenv-fake-gui=true`: The seed for the RNG governing the operation of the fake GUI component. This is entirely independent of the RNGs used by the model.

cmdenv-interactive = <*bool*>, default: `false`
Per-simulation-run setting.
Defines what Cmdenv should do when the model contains unassigned parameters. In interactive mode, it asks the user. In non-interactive mode (which is more suitable for batch execution), Cmdenv stops with an error.

****.cmdenv-log-level** = <*string*>, default: `TRACE`
Per-object setting for modules.
Specifies the per-component level of detail recorded by log statements, output below the specified level is omitted. Available values are (case insensitive): `off`, `fatal`, `error`, `warn`, `info`, `detail`, `debug` or `trace`. Note that the level of detail is also controlled by the globally specified runtime log level and the `COMPILETIME_LOGLEVEL` macro that is used to completely remove log statements from the executable.

cmdenv-log-prefix = <*string*>, default: `[%l]`
Per-simulation-run setting.
Specifies the format string that determines the prefix of each log line. The format string may contain format directives in the syntax `%x` (a `%` followed by a single format character). For example `%l` stands for log level, and `%J` for source component. See the manual for the list of available format characters.

cmdenv-output-file = <*filename*>, default: `${resultdir}/${configname}-${iterationvarsf}#${repetition}.out`
Per-simulation-run setting.
When `cmdenv-record-output=true`: file name to redirect standard output to. See also `fname-append-host`.

cmdenv-performance-display = <*bool*>, default: `true`
Per-simulation-run setting.
When `cmdenv-express-mode=true`: print detailed performance information. Turning it on results in a 3-line entry printed on each update, containing ev/sec, simsec/sec, ev/simsec, number of messages created/still present/currently scheduled in FES.

cmdenv-redirect-output = <*bool*>, default: `false`
Per-simulation-run setting.
Causes Cmdenv to redirect standard output of simulation runs to a file or separate files per run. This option can be useful with running simulation campaigns (e.g. using opp_runall), and also with parallel simulation. See also: `cmdenv-output-file`, `fname-append-host`.

cmdenv-runs-to-execute = <*string*>
Global setting (applies to all simulation runs).
Specifies which runs to execute from the selected configuration (see `cmdenv-config-name` option). It accepts a filter expression of iteration variables such as `$numHosts>10`

`&& $iatime==1s`, or a comma-separated list of run numbers or run number ranges, e.g. `1,3..4,7..9`. If the value is missing, Cmdenv executes all runs in the selected configuration. The `-r` command line option overrides this setting.

cmdenv-status-frequency = <*double*>, unit=`s`, default: `2s`
Per-simulation-run setting.
When `cmdenv-express-mode=true`: print status update every n seconds.

cmdenv-stop-batch-on-error = <*bool*>, default: `true`
Per-simulation-run setting.
Decides whether Cmdenv should skip the rest of the runs when an error occurs during the execution of one run.

config-recording = <*custom*>, default: `all`
Per-simulation-run setting.
Selects the set of config options to save into result files. This option can help reduce the size of result files, which is especially useful in the case of large simulation campaigns. Possible values: all, none, config, params, essentials, globalconfig

configuration-class = <*string*>
Global setting (applies to all simulation runs).
Part of the Envir plugin mechanism: selects the class from which all configuration information will be obtained. This option lets you replace omnetpp.ini with some other implementation, e.g. database input. The simulation program still has to bootstrap from an omnetpp.ini (which contains the configuration-class setting). The class should implement the `cConfigurationEx` interface.

constraint = <*string*>
Per-simulation-run setting.
For scenarios. Contains an expression that iteration variables (`${}` syntax) must satisfy for that simulation to run. Example: `$i < $j+1`.

cpu-time-limit = <*double*>, unit=`s`
Per-simulation-run setting.
Stops the simulation when CPU usage has reached the given limit. The default is no limit. Note: To reduce per-event overhead, this time limit is only checked every N events (by default, N=1024).

debug-on-errors = <*bool*>, default: `false`
Per-simulation-run setting.
When set to true, runtime errors will cause the simulation program to break into the C++ debugger (if the simulation is running under one, or just-in-time debugging is activated). Once in the debugger, you can view the stack trace or examine variables.

debug-statistics-recording = <*bool*>, default: `false`
Per-simulation-run setting.
Turns on the printing of debugging information related to statistics recording (`@statistic` properties)

debugger-attach-command = <*string*>
Global setting (applies to all simulation runs).
The command line to launch the debugger. It must contain exactly one percent sign, as `%u`, which will be replaced by the PID of this process. The command must not block (i.e. it should end in `&` on Unix-like systems). It will be executed by the default system shell (on Windows, usually cmd.exe). Default on this platform: `opp_ide`

`omnetpp://cdt/debugger/attach?pid=%u`. This default can be overridden with the `OMNETPP_DEBUGGER_COMMAND` environment variable.

debugger-attach-on-error = <*bool*>, default: `false`
Global setting (applies to all simulation runs).
When set to true, runtime errors and crashes will trigger an external debugger to be launched (if not already present), allowing you to perform just-in-time debugging on the simulation process. The debugger command is configurable. Note that debugging (i.e. attaching to) a non-child process needs to be explicitly enabled on some systems, e.g. Ubuntu.

debugger-attach-on-startup = <*bool*>, default: `false`
Global setting (applies to all simulation runs).
When set to true, the simulation program will launch an external debugger attached to it (if not already present), allowing you to set breakpoints before proceeding. The debugger command is configurable. Note that debugging (i.e. attaching to) a non-child process needs to be explicitly enabled on some systems, e.g. Ubuntu.

debugger-attach-wait-time = <*double*>, unit=`s`, default: `20s`
Global setting (applies to all simulation runs).
An interval to wait after launching the external debugger, to give the debugger time to start up and attach to the simulation process.

description = <*string*>
Per-simulation-run setting.
Descriptive name for the given simulation configuration. Descriptions get displayed in the run selection dialog.

..display-name** = <*string*>
Per-object setting for modules.
Specifies a display name for the module, which is shown e.g. in Qtenv's graphical module view.

..display-string** = <*string*>
Per-object setting for modules and channels.
Additional display string for the module/channel; it will be merged into the display string given via `@display` properties, and override its content.

eventlog-file = <*filename*>, default: `${resultdir}/${configname}-${iterationvarsf}#${repetition}.elog`
Per-simulation-run setting.
Name of the eventlog file to generate.

eventlog-index-frequency = <*double*>, unit=`B`, default: `1 MiB`
Per-simulation-run setting.
The eventlog file contains incremental snapshots called index. An index is much smaller than a full snapshot, but it only contains the differences since the last index.

eventlog-max-size = <*double*>, unit=`B`, default: `10 GiB`
Per-simulation-run setting.
Specify the maximum size of the eventlog file in bytes. The eventlog file is automatically truncated when this limit is reached.

eventlog-message-detail-pattern = <*custom*>
Per-simulation-run setting.
A list of patterns separated by '|' character which will be used to write message detail

information into the eventlog for each message sent during the simulation. The message detail will be presented in the sequence chart tool. Each pattern starts with an object pattern optionally followed by ':' character and a comma separated list of field patterns. In both patterns and/or/not/* and various field match expressions can be used. The object pattern matches to class name, the field pattern matches to field name by default.

```
EVENTLOG-MESSAGE-DETAIL-PATTERN := ( DETAIL-PATTERN '|' )* DETAIL_PATTERN
DETAIL-PATTERN := OBJECT-PATTERN [ ':'  FIELD-PATTERNS ]
OBJECT-PATTERN := MATCH-EXPRESSION
FIELD-PATTERNS := ( FIELD-PATTERN ',' )* FIELD_PATTERN
FIELD-PATTERN := MATCH-EXPRESSION
```

Examples:
`*`: captures all fields of all messages
`*Frame:*Address,*Id`: captures all fields named somethingAddress and somethingId from messages of any class named somethingFrame

`MyMessage:declaredOn=~MyMessage`: captures instances of MyMessage recording the fields declared on the MyMessage class
`*:(not declaredOn=~cMessage and not declaredOn=~cNamedObject and not declaredOn=~cObject)`: records user-defined fields from all messages

eventlog-min-truncated-size = <*double*>, unit=`B`, default: `1 GiB`
Per-simulation-run setting.
Specify the minimum size of the eventlog file in bytes after the file is truncated. Truncation means older events are discarded while newer ones are kept.

eventlog-options = <*custom*>
Per-simulation-run setting.
The content of the eventlog is diveded into categories. This option allows to record only certain categories reducing the file size. Specify a comma separated subset of the following keywords: text, message, module, methodcall, displaystring and custom. By default all categories are enabled.

eventlog-recording-intervals = <*custom*>
Per-simulation-run setting.
Simulation time interval(s) when events should be recorded. Syntax: `[<from>]..[<to>]`, `...` That is, both start and end of an interval are optional, and intervals are separated by comma. Example: `..10.2, 22.2..100, 233.3..`

eventlog-snapshot-frequency = <*double*>, unit=`B`, default: `100 MiB`
Per-simulation-run setting.
The eventlog file contains snapshots periodically. Each one describes the complete simulation state at a specific event. Snapshots help various tools to handle large eventlog files more efficiently. Specifying greater value means less help, while smaller value means bigger eventlog files.

eventlogmanager-class = <*string*>, default: `omnetpp::envir::EventlogFileManager`
Per-simulation-run setting.
Part of the Envir plugin mechanism: selects the eventlog manager class to be used to record data. The class has to implement the `cIEventlogManager` interface.

experiment-label = <*string*>, default: `${configname}`
Per-simulation-run setting.

Identifies the simulation experiment (which consists of several, potentially repeated measurements). This string gets recorded into result files, and may be referred to during result analysis.

extends = <*string*>
Per-simulation-run setting.
Name of the configuration this section is based on. Entries from that section will be inherited and can be overridden. In other words, configuration lookups will fall back to the base section.

fingerprint = <*string*>
Per-simulation-run setting.
The expected fingerprints of the simulation. If you need multiple fingerprints, separate them with commas. When provided, the fingerprints will be calculated from the specified properties of simulation events, messages, and statistics during execution, and checked against the provided values. Fingerprints are suitable for crude regression tests. As fingerprints occasionally differ across platforms, more than one value can be specified for a single fingerprint, separated by spaces, and a match with any of them will be accepted. To obtain a fingerprint, enter a dummy value (such as `0000`), and run the simulation.

fingerprint-events = <*string*>, default: `*`
Per-simulation-run setting.
Configures the fingerprint calculator to consider only certain events. The value is a pattern that will be matched against the event name by default. It may also be an expression containing pattern matching characters, field access, and logical operators. The default setting is '*' which includes all events in the calculated fingerprint. If you configured multiple fingerprints, separate filters with commas.

fingerprint-ingredients = <*string*>, default: `tplx`
Per-simulation-run setting.
Specifies the list of ingredients to be taken into account for fingerprint computation. Each character corresponds to one ingredient: 'e' event number, 't' simulation time, 'n' message (event) full name, 'c' message (event) class name, 'k' message kind, 'l' message bit length, 'o' message control info class name, 'd' message data, 'i' module id, 'm' module full name, 'p' module full path, 'a' module class name, 'r' random numbers drawn, 's' scalar results, 'z' statistic results, 'v' vector results, 'x' extra data provided by modules. Note: ingredients specified in an expected fingerprint (characters after the '/' in the fingerprint value) take precedence over this setting. If you configured multiple fingerprints, separate ingredients with commas.

fingerprint-modules = <*string*>, default: `*`
Per-simulation-run setting.
Configures the fingerprint calculator to consider only certain modules. The value is a pattern that will be matched against the module full path by default. It may also be an expression containing pattern matching characters, field access, and logical operators. The default setting is '*' which includes all events in all modules in the calculated fingerprint. If you configured multiple fingerprints, separate filters with commas.

fingerprint-results = <*string*>, default: `*`
Per-simulation-run setting.
Configures the fingerprint calculator to consider only certain results. The value is a pattern that will be matched against the result full path by default. It may also be an expression containing pattern matching characters, field access, and logical operators. The

default setting is '*' which includes all results in all modules in the calculated fingerprint. If you configured multiple fingerprints, separate filters with commas.

fingerprintcalculator-class = <*string*>, default: `omnetpp::cSingleFingerprintCalculator`
Per-simulation-run setting.
Part of the Envir plugin mechanism: selects the fingerprint calculator class to be used to calculate the simulation fingerprint. The class has to implement the `cFingerprintCalculator` interface.

fname-append-host = <*bool*>
Global setting (applies to all simulation runs).
Turning it on will cause the host name and process Id to be appended to the names of output files (e.g. omnetpp.vec, omnetpp.sca). This is especially useful with distributed simulation. The default value is true if parallel simulation is enabled, false otherwise.

futureeventset-class = <*string*>, default: `omnetpp::cEventHeap`
Per-simulation-run setting.
Part of the Envir plugin mechanism: selects the class for storing the future events in the simulation. The class has to implement the `cFutureEventSet` interface.

image-path = <*path*>, default: `./images`
Global setting (applies to all simulation runs).
A semicolon-separated list of directories that contain module icons and other resources. This list will be concatenated with the contents of the `OMNETPP_IMAGE_PATH` environment variable or with a compile-time, hardcoded image path if the environment variable is empty.

iteration-nesting-order = <*string*>
Per-simulation-run setting.
Specifies the loop nesting order for iteration variables (`${}` syntax). The value is a comma-separated list of iteration variables; the list may also contain at most one asterisk. Variables that are not explicitly listed will be inserted at the position of the asterisk, or appended to the list if there is no asterisk. The first variable will become the outermost loop, and the last one the innermost loop. Example: `repetition,numHosts,*,iaTime`.

load-libs = <*filenames*>
Global setting (applies to all simulation runs).
A space-separated list of dynamic libraries to be loaded on startup. The libraries should be given without the `.dll` or `.so` suffix – that will be automatically appended.

max-module-nesting = <*int*>, default: `50`
Per-simulation-run setting.
The maximum allowed depth of submodule nesting. This is used to catch accidental infinite recursions in NED.

measurement-label = <*string*>, default: `${iterationvars}`
Per-simulation-run setting.
Identifies the measurement within the experiment. This string gets recorded into result files, and may be referred to during result analysis.

****.module-eventlog-recording** = <*bool*>, default: `true`
Per-object setting for simple modules.
Enables recording events on a per module basis. This is meaningful for simple modules only. Usage: `<module-full-path>.module-eventlog-recording=true/false`. Examples: `**.router[10..20].**.module-eventlog-recording = true`; `**.module-eventlog-recording = false`

ned-package-exclusions = <*custom*>
Global setting (applies to all simulation runs).
A semicolon-separated list of NED packages to be excluded when loading NED files. Sub-packages of excluded ones are also excluded. Additional items may be specified via the `-x` command-line option and the `OMNETPP_NED_PACKAGE_EXCLUSIONS` environment variable.

ned-path = <*path*>
Global setting (applies to all simulation runs).
A semicolon-separated list of directories. The directories will be regarded as roots of the NED package hierarchy, and all NED files will be loaded from their subdirectory trees. This option is normally left empty, as the OMNeT++ IDE sets the NED path automatically, and for simulations started outside the IDE it is more convenient to specify it via command-line option (-n) or via environment variable (OMNETPP_NED_PATH, NEDPATH).

network = <*string*>
Per-simulation-run setting.
The name of the network to be simulated. The package name can be omitted if the ini file is in the same directory as the NED file that contains the network.

num-rngs = <*int*>, default: `1`
Per-simulation-run setting.
The number of random number generators.

output-scalar-db-commit-freq = <*int*>, default: `100000`
Global setting (applies to all simulation runs).
Used with SqliteOutputScalarManager: COMMIT every n INSERTs.

output-scalar-file = <*filename*>, default: `${resultdir}/${configname}-${iterationvarsf}#${repetition}.sca`
Per-simulation-run setting.
Name for the output scalar file.

output-scalar-file-append = <*bool*>, default: `false`
Per-simulation-run setting.
What to do when the output scalar file already exists: append to it (OMNeT++ 3.x behavior), or delete it and begin a new file (default).

output-scalar-precision = <*int*>, default: `14`
Per-simulation-run setting.
The number of significant digits for recording data into the output scalar file. The maximum value is ~15 (IEEE double precision). This has no effect on SQLite recording, as it stores values as 8-byte IEEE floating point numbers.

output-vector-db-indexing = <*custom*>, default: `skip`
Global setting (applies to all simulation runs).
Whether and when to add an index to the 'vectordata' table in SQLite output vector files. Possible values: skip, ahead, after

output-vector-file = <*filename*>, default: `${resultdir}/${configname}-${iterationvarsf}#${repetition}.vec`
Per-simulation-run setting.
Name for the output vector file.

output-vector-file-append = <*bool*>, default: `false`
Per-simulation-run setting.
What to do when the output vector file already exists: append to it, or delete it and begin a new file (default). Note: `cIndexedFileOutputVectorManager` currently does not support appending.

output-vector-precision = <*int*>, default: `14`
Per-simulation-run setting.
The number of significant digits for recording data into the output vector file. The maximum value is ~15 (IEEE double precision). This setting has no effect on SQLite recording (it stores values as 8-byte IEEE floating point numbers), and for the "time" column which is represented as fixed-point numbers and always get recorded precisely.

output-vectors-memory-limit = <*double*>, unit=`B`, default: `16MiB`
Per-simulation-run setting.
Total memory that can be used for buffering output vectors. Larger values produce less fragmented vector files (i.e. cause vector data to be grouped into larger chunks), and therefore allow more efficient processing later. There is also a per-vector limit, see `**.vector-buffer`.

outputscalarmanager-class = <*string*>, default: `omnetpp::envir::OmnetppOutputScalarManager`
Per-simulation-run setting.
Part of the Envir plugin mechanism: selects the output scalar manager class to be used to record data passed to recordScalar(). The class has to implement the `cIOutputScalarManager` interface.

outputvectormanager-class = <*string*>, default: `omnetpp::envir::OmnetppOutputVectorManager`
Per-simulation-run setting.
Part of the Envir plugin mechanism: selects the output vector manager class to be used to record data from output vectors. The class has to implement the `cIOutputVectorManager` interface.

parallel-simulation = <*bool*>, default: `false`
Global setting (applies to all simulation runs).
Enables parallel distributed simulation.

****.param-record-as-scalar** = <*bool*>, default: `false`
Per-object setting for module/channel parameters.
Applicable to module parameters: specifies whether the module parameter should be recorded into the output scalar file. Set it for parameters whose value you will need for result analysis.

****.param-recording** = <*bool*>, default: `true`
Per-object setting for module/channel parameters.
Whether the matching module (and channel) parameters should be recorded.
Usage: `<module-full-path>.<parameter-name>.param-recording=true/false`.
Example: `**.app.pkLen.param-recording=true`

parameter-mutability-check = <*bool*>, default: `true`
Per-simulation-run setting.
Setting to false will disable errors raised when trying to change the values of module/channel parameters not marked as @mutable. This is primarily a compatibility setting intended to facilitate running simulation models that were not yet annotated with @mutable.

parsim-communications-class = <*string*>, default: `omnetpp::cFileCommunications`
Global setting (applies to all simulation runs).
If `parallel-simulation=true`, it selects the class that implements communication between partitions. The class must implement the `cParsimCommunications` interface.

parsim-debug = <*bool*>, default: `true`
Global setting (applies to all simulation runs).
With `parallel-simulation=true`: turns on printing of log messages from the parallel simulation code.

parsim-filecommunications-prefix = <*string*>, default: `comm/`
Global setting (applies to all simulation runs).
When `cFileCommunications` is selected as parsim communications class: specifies the prefix (directory+potential filename prefix) for creating the files for cross-partition messages.

parsim-filecommunications-preserve-read = <*bool*>, default: `false`
Global setting (applies to all simulation runs).
When `cFileCommunications` is selected as parsim communications class: specifies that consumed files should be moved into another directory instead of being deleted.

parsim-filecommunications-read-prefix = <*string*>, default: `comm/read/`
Global setting (applies to all simulation runs).
When `cFileCommunications` is selected as parsim communications class: specifies the prefix (directory) where files will be moved after having been consumed.

parsim-idealsimulationprotocol-tablesize = <*int*>, default: `100000`
Global setting (applies to all simulation runs).
When `cIdealSimulationProtocol` is selected as parsim synchronization class: specifies the memory buffer size for reading the ISP event trace file.

parsim-mpicommunications-mpibuffer = <*int*>
Global setting (applies to all simulation runs).
When `cMPICommunications` is selected as parsim communications class: specifies the size of the MPI communications buffer. The default is to calculate a buffer size based on the number of partitions.

parsim-namedpipecommunications-prefix = <*string*>, default: `comm/`
Global setting (applies to all simulation runs).
When `cNamedPipeCommunications` is selected as parsim communications class: selects the prefix (directory+potential filename prefix) where name pipes are created in the file system.

parsim-nullmessageprotocol-laziness = <*double*>, default: `0.5`
Global setting (applies to all simulation runs).
When `cNullMessageProtocol` is selected as parsim synchronization class: specifies the laziness of sending null messages. Values in the range `[0,1)` are accepted. Laziness=0 causes null messages to be sent out immediately as a new EOT is learned, which may result in excessive null message traffic.

parsim-nullmessageprotocol-lookahead-class = <*string*>, default: `cLinkDelayLookahead`
Global setting (applies to all simulation runs).
When `cNullMessageProtocol` is selected as parsim synchronization class: specifies the C++ class that calculates lookahead. The class should subclass from `cNMPLookahead`.

parsim-num-partitions = <*int*>
Global setting (applies to all simulation runs).
If `parallel-simulation=true`, it tells the number of parallel processes to use. This value must be in agreement with the number of simulator instances launched, e.g. with the `-n` or `-np` command-line option specified to the `mpirun` program.

parsim-synchronization-class = <*string*>, default: `omnetpp::cNullMessageProtocol`
Global setting (applies to all simulation runs).
If `parallel-simulation=true`, it selects the parallel simulation algorithm. The class must implement the `cParsimSynchronizer` interface.

****.partition-id** = <*string*>
Per-object setting for modules.
With parallel simulation: in which partition the module should be instantiated. Specify numeric partition ID, or a comma-separated list of partition IDs for compound modules that span across multiple partitions. Ranges (`5..9`) and `*` (=all) are accepted too.

print-undisposed = <*bool*>, default: `true`
Per-simulation-run setting.
Whether to report objects left (that is, not deallocated by simple module destructors) after network cleanup.

qtenv-default-config = <*string*>
Global setting (applies to all simulation runs).
Specifies which config Qtenv should set up automatically on startup. The default is to ask the user.

qtenv-default-run = <*string*>
Global setting (applies to all simulation runs).
Specifies which run (of the default config, see `qtenv-default-config`) Qtenv should set up automatically on startup. A run filter is also accepted. The default is to ask the user.

qtenv-extra-stack = <*double*>, unit=`B`, default: `80KiB`
Global setting (applies to all simulation runs).
Specifies the extra amount of stack that is reserved for each `activity()` simple module when the simulation is run under Qtenv.

real-time-limit = <*double*>, unit=`s`
Per-simulation-run setting.
Stops the simulation after the specified amount of time has elapsed. The default is no limit. Note: To reduce per-event overhead, this time limit is only checked every N events (by default, N=1024).

realtimescheduler-scaling = <*double*>
Global setting (applies to all simulation runs).
When cRealTimeScheduler is selected as scheduler class: ratio of simulation time to real time. For example, `realtimescheduler-scaling=2` will cause simulation time to progress twice as fast as runtime.

record-eventlog = <*bool*>, default: `false`
Per-simulation-run setting.
Enables recording an eventlog file, which can be later visualized on a sequence chart. See `eventlog-file` option too.

repeat = <*int*>, default: `1`
Per-simulation-run setting.
For scenarios. Specifies how many replications should be done with the same parameters (iteration variables). This is typically used to perform multiple runs with different random number seeds. The loop variable is available as `${repetition}`. See also: `seed-set` key.

replication-label = <*string*>, default: `#${repetition}`
Per-simulation-run setting.
Identifies one replication of a measurement (see `repeat` and `measurement-label` options as well). This string gets recorded into result files, and may be referred to during result analysis.

result-dir = <*string*>, default: `results`
Per-simulation-run setting.
Base value for the `${resultdir}` variable, which is used as the default directory for result files (output vector file, output scalar file, eventlog file, etc.). See also the `resultdir-subdivision` config option.

****.result-recording-modes** = <*string*>, default: `default`
Per-object setting for statistics (@statistic).
Defines how to calculate results from the matching `@statistic`.
Usage: `<module-full-path>.<statistic-name>.result-recording-modes=<modes>`.
Special values: `default`, `all`: they select the modes listed in the `record` key of `@statistic`; all selects all of them, default selects the non-optional ones (i.e. excludes the ones that end in a question mark). Example values: `vector`, `count`, `last`, `sum`, `mean`, `min`, `max`, `timeavg`, `stats`, `histogram`. More than one values are accepted, separated by commas. Expressions are allowed. Items prefixed with `-` get removed from the list. Example: `**.queueLength.result-recording-modes=default,-vector,+timeavg`

resultdir-subdivision = <*bool*>, default: `false`
Per-simulation-run setting.
Makes the results directory hierarchical by appending `${iterationvarsd}` to the value of the `result-dir` config option. This is useful if a parameter study produces a large number of runs (>10000), as many file managers and other tools (including the OMNeT++ IDE) struggle with directories containing that many files. An alternative to using this option is to include iteration variables directly in the value of the `result-dir` option.

****.rng-%** = <*int*>
Per-object setting for modules and channels.
Maps a module-local RNG to one of the global RNGs. Example: `**.gen.rng-1=3` maps the local RNG 1 of modules matching `**.gen` to the global RNG 3. The value may be an expression, with the `index` and `ancestorIndex()` operators being potentially very useful. The default is one-to-one mapping, i.e. RNG k of all modules refer to the global RNG k (`for k=0..num-rngs-1`).
Usage: `<module-full-path>.rng-<local-index>=<global-index>`. Examples: `**.mac.rng-0=1; **.source[*].rng-0=index`

rng-class = <*string*>, default: `omnetpp::cMersenneTwister`
Per-simulation-run setting.
The random number generator class to be used. It can be `cMersenneTwister`, `cLCG32`, `cAkaroaRNG`, or you can use your own RNG class (it must be subclassed from `cRNG`).

runnumber-width = <*int*>, default: `0`
Per-simulation-run setting.

Setting a nonzero value will cause the `$runnumber` variable to get padded with leading zeroes to the given length.

****.scalar-recording** = <*bool*>, default: `true`
Per-object setting for scalar results.
Whether the matching output scalars and statistic objects should be recorded.
Usage: `<module-full-path>.<scalar-name>.scalar-recording=true/false`. To enable/disable individual recording modes for a @statistic (those added via the `record=...` key of `@statistic` or the `**.result-recording-modes=...` config option), use `<statistic-name>:<mode>` for `<scalar-name>`, and make sure the `@statistic` as a whole is not disabled with `**.<statistic-name>.statistic-recording=false`.
Example: `**.ping.roundTripTime:stddev.scalar-recording=false`

scheduler-class = <*string*>, default: `omnetpp::cSequentialScheduler`
Per-simulation-run setting.
Part of the Envir plugin mechanism: selects the scheduler class. This plugin interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation. The class has to implement the `cScheduler` interface.

sectionbasedconfig-configreader-class = <*string*>
Global setting (applies to all simulation runs).
When `configuration-class=SectionBasedConfiguration`: selects the configuration reader C++ class, which must subclass from `cConfigurationReader`.

seed-%-lcg32 = <*int*>
Per-simulation-run setting.
When cLCG32 is selected as random number generator: seed for the kth RNG. (Substitute k for '%' in the key.)

seed-%-mt = <*int*>
Per-simulation-run setting.
When Mersenne Twister is selected as random number generator (default): seed for RNG number k. (Substitute k for '%' in the key.)

seed-%-mt-p% = <*int*>
Per-simulation-run setting.
With parallel simulation: When Mersenne Twister is selected as random number generator (default): seed for RNG number k in partition number p. (Substitute k for the first '%' in the key, and p for the second.)

seed-set = <*int*>, default: `${runnumber}`
Per-simulation-run setting.
Selects the kth set of automatic random number seeds for the simulation. Meaningful values include `${repetition}` which is the repeat loop counter (see `repeat` option), and `${runnumber}`.

sim-time-limit = <*double*>, unit=`s`
Per-simulation-run setting.
Stops the simulation when simulation time reaches the given limit. The default is no limit.

simtime-resolution = <*custom*>, default: `ps`
Global setting (applies to all simulation runs).
Sets the resolution for the 64-bit fixed-point simulation time representation. Accepted values are: second-or-smaller time units (`s`, `ms`, `us`, `ns`, `ps`, `fs` or as), power-of-ten

multiples of such units (e.g. 100ms), and base-10 scale exponents in the -18..0 range. The maximum representable simulation time depends on the resolution. The default is picosecond resolution, which offers a range of ~110 days.

simtime-scale = <*int*>, default: `-12`
Global setting (applies to all simulation runs).
DEPRECATED in favor of simtime-resolution. Sets the scale exponent, and thus the resolution of time for the 64-bit fixed-point simulation time representation. Accepted values are -18..0; for example, -6 selects microsecond resolution. -12 means picosecond resolution, with a maximum simtime of ~110 days.

snapshot-file = <*filename*>, default: `${resultdir}/${configname}-${iterationvarsf}#${repetition}.sna`
Per-simulation-run setting.
Name of the snapshot file.

snapshotmanager-class = <*string*>, default: `omnetpp::envir::FileSnapshotManager`
Per-simulation-run setting.
Part of the Envir plugin mechanism: selects the class to handle streams to which snapshot() writes its output. The class has to implement the `cISnapshotManager` interface.

..statistic-recording** = <*bool*>, default: `true`
Per-object setting for statistics (@statistic).
Whether the matching `@statistic` should be recorded. This option lets one completely disable all recording from a @statistic. Disabling a `@statistic` this way is more efficient than specifying `**.scalar-recording=false` and `**.vector-recording=false` together.
Usage: `<module-full-path>.<statistic-name>.statistic-recording=true/false`.
Example: `**.ping.roundTripTime.statistic-recording=false`

total-stack = <*double*>, unit=`B`
Global setting (applies to all simulation runs).
Specifies the maximum memory for `activity()` simple module stacks. You need to increase this value if you get a "Cannot allocate coroutine stack" error.

..typename** = <*string*>
Per-object setting for modules and channels.
Specifies type for submodules and channels declared with 'like <>'.

user-interface = <*string*>
Global setting (applies to all simulation runs).
Selects the user interface to be started. Known good values are Cmdenv and Qtenv. This option is normally left empty, as it is more convenient to specify the user interface via a command-line option or the IDE's Run and Debug dialogs. New user interfaces can be defined by subclassing `cRunnableEnvir`.

..vector-buffer** = <*double*>, unit=`B`, default: `1MiB`
Per-object setting for vector results.
For output vectors: the maximum per-vector buffer space used for storing values before writing them out as a block into the output vector file. There is also a total limit, see `output-vectors-memory-limit`.
Usage: `<module-full-path>.<vector-name>.vector-buffer=<amount>`.

..vector-record-eventnumbers** = <*bool*>, default: `true`
Per-object setting for vector results.

Whether to record event numbers for an output vector. (Values and timestamps are always recorded.) Event numbers are needed by the Sequence Chart Tool, for example. Usage: `<module-full-path>.<vector-name>.vector-record-eventnumbers=true/false`. Example: `**.ping.roundTripTime:vector.vector-record-eventnumbers=false`

****.vector-recording** = <*bool*>, default: `true`
Per-object setting for vector results.
Whether data written into an output vector should be recorded. Usage: `<module-full-path>.<vector-name>.vector-recording=true/false`. To control vector recording from a `@statistic`, use `<statistic-name>:vector for <vector-name>`. Example: `**.ping.roundTripTime:vector.vector-recording=false`

****.vector-recording-intervals** = <*custom*>
Per-object setting for vector results.
Allows one to restrict recording of an output vector to one or more simulation time intervals. Usage: `<module-full-path>.<vector-name>.vector-recording-intervals=<intervals>`. The syntax for `<intervals>` is: `[<from>]..[<to>],...` That is, both start and end of an interval are optional, and intervals are separated by comma. Example: `**.roundTripTime:vector.vector-recording-intervals=..100, 200..400, 900..`

warmup-period = <*double*>, unit=`s`
Per-simulation-run setting.
Length of the initial warm-up period. When set, results belonging to the first x seconds of the simulation will not be recorded into output vectors, and will not be counted into output scalars (see option `**.result-recording-modes`). This option is useful for steady-state simulations. The default is 0s (no warmup period). Note that models that compute and record scalar results manually (via `recordScalar()`) will not automatically obey this setting.

warnings = <*bool*>, default: `true`
Per-simulation-run setting.
Enables warnings.

I.2 Predefined Variables

Predefined variables that can be used in config values:

${runid} :
A reasonably globally unique identifier for the run, produced by concatenating the configuration name, run number, date/time, etc.

${inifile} :
Name of the (primary) inifile

${configname} :
Name of the active configuration

${runnumber} :
Sequence number of the current run within all runs in the active configuration

${network} :
Value of the `network` configuration option

${experiment} :
Value of the `experiment-label` configuration option

${measurement} :
Value of the `measurement-label` configuration option

${replication} :
Value of the `replication-label` configuration option

${processid} :
PID of the simulation process

${datetime} :
Date and time the simulation run was started

${datetimef} :
Like ${datetime}, but sanitized for use as part of a file name

${resultdir} :
Value of the `result-dir` configuration option

${repetition} :
The iteration number in `0..N-1`, where `N` is the value of the `repeat` configuration option

${seedset} :
Value of the `seed-set` configuration option

${iterationvars} :
Concatenation of all user-defined iteration variables in `name=value` form

${iterationvarsf} :
Like ${iterationvars}, but sanitized for use as part of a file name

${iterationvarsd} :
Like ${iterationvars}, but for use as hierarchical folder name (it contains slashes where ${iterationvarsf} has commas)

Appendix J

Result File Formats

J.1 Native Result Files

The file format described here applies to *both output vector and output scalar files*. Their formats are consistent, only the types of entries occurring in them are different. This unified format also means that they can be read with a common routine.

Result files are *line oriented*. A line consists of one or more tokens, separated by whitespace. Tokens either do not contain whitespace, or whitespace is escaped using a backslash, or are quoted using double quotes. Escaping within quotes using backslashes is also permitted.

The first token of a line usually identifies the type of the entry. A notable exception is an output vector data line, which begins with a numeric identifier of the given output vector.

A line starting with # as the first non-whitespace character denotes a comment, and is to be ignored during processing.

Result files are written from simulation runs. A simulation run generates physically contiguous sets of lines into one or more result files. (That is, lines from different runs do not arbitrarily mix in the files.)

A run is identified by a unique textual *runId*, which appears in all result files written during that run. The runId may appear on the user interface, so it should be somewhat meaningful to the user. Nothing should be assumed about the particular format of runId, but it will be some string concatenated from the simulated network's name, the time/date, the hostname, and other pieces of data to make it unique.

A simulation run will typically write into two result files (.vec and .sca). However, when using parallel distributed simulation, the user will end up with several .vec and .sca files, because different partitions (a separate process each) will write into different files. However, all these files will contain the same runId, so it is possible to relate data that belong together.

Entry types are:

- **version**: result file version
- **run**: simulation run identifier
- **attr**: run, vector, scalar or statistics object attribute
- **itervar**: iteration variable

- **config**: configuration entry
- **par**: module parameter
- **scalar**: scalar data
- **vector**: vector declaration
- *vector-id*: vector data
- **file**: vector file attributes
- **statistic**: statistics object
- **field**: field of a statistics object
- **bin**: histogram bin

J.1.1 Version

Specifies the format of the result file. It is written at the beginning of the file.

Syntax:

version *versionNumber*

The version described in this document is 3, used since OMNeT++ 6.0. Version 1 files were produced by OMNeT++ 3.x and earlier, and version 2 files by OMNeT++ 4.x and 5.x.[1]

J.1.2 Run Declaration

Marks the beginning of a new run in the file. Entries after this line belong to this run.

Syntax:

run *runId*

Example:

```
run TokenRing1-0-20080514-18:19:44-3248
```

Typically there will be one run per file, but this is not mandatory. In cases when there are more than one run in a file and it is not feasible to keep the entire file in memory during analysis, the offsets of the *run* lines may be indexed for more efficient random access.

The *run* line may be immediately followed by *attribute* lines. Attributes may store generic data like the network name, date/time of running the simulation, configuration options that took effect for the simulation, etc.

Run attribute names used by OMNEST include the following:

Generic attribute names:

[1] Differences between version 2 and version 3 files are minimal, and mostly only affect the run header. Version 3 introduced *itervar* lines to allow distinguishing iteration variables from other run attributes (in version 2 they were all recorded in *attr* lines). *param* lines in version 2 (which recorded parameter assignment entries in the configuration) have been replaced in version 3 with the more general *config* lines (which record all configuration entries, not just parameter assignments). In version 2, parameter values (if requested) were recorded as scalars, whereas in version 3 they are recorded in *par* lines, which allow recording of volatile parameters (as expressions) and non-numeric values as well. Additionally, version 3 doesn't record the fields *sum* and *sqrsum* for weighted statistics.

- **network**: name of the network simulated
- **datetime**: date/time associated with the run
- **processid**: the PID of the simulation process
- **inifile**: the main configuration file
- **configname**: name of the inifile configuration
- **seedset**: index of the seed-set use for the simulation

Attributes associated with parameter studies (iterated runs):

- **runnumber**: the run number within the parameter study
- **experiment**: experiment label
- **measurement**: measurement label
- **replication**: replication label
- **repetition**: the loop counter for repetitions with different seeds
- **iterationvars**: string containing the values of the iteration variables
- **iterationvarsf**: like `iterationvars`, but sanitized for use as part of file names

An example run header:

```
run PureAlohaExperiment-0-20200304-18:05:49-194559
attr configname PureAlohaExperiment
attr datetime 20200304-18:05:49
attr experiment PureAlohaExperiment
attr inifile omnetpp.ini
attr iterationvars "$numHosts=10, $iaMean=1"
attr measurement "$numHosts=10, $iaMean=1"
attr network Aloha
attr processid 194559
attr repetition 0
attr replication #0
attr resultdir results
attr runnumber 0
attr seedset 0
itervar iaMean 1
itervar numHosts 10
config repeat 2
config sim-time-limit 90min
config network Aloha
config Aloha.numHosts 10
config Aloha.host[*].iaTime exponential(1s)
config Aloha.numHosts 20
config Aloha.txRate 9.6kbps
config **.x "uniform(0m, 1000m)"
config **.y "uniform(0m, 1000m)"
config **.idleAnimationSpeed 1
```

J.1.3 Attributes

Contains an attribute for the preceding run, vector, scalar or statistics object. Attributes can be used for saving arbitrary extra information for objects; processors should ignore unrecognized attributes.

Syntax:

attr *name value*

Example:

```
attr network "largeNet"
```

J.1.4 Iteration Variables

Contains an iteration variable for the given run.

Syntax:

itervar *name value*

Examples:

```
itervar numHosts 10
itervar tcpType Reno
```

J.1.5 Configuration Entries

The configuration of the simulation is captured in the result file as an ordered list of *config* lines. The list contains both the contents of ini files and the options given one the command line.

The order of lines represents a flattened view of the ini file(s). The contents of sections are recorded in an order that reflects the section inheritance graph: derived sections precede the sections they extend (so `General` comes last), and the contents of unrelated sections are omitted. Command like options are at the top. The relative order of lines within ini file sections are preserved. This order corresponds to the search order of entries that contain wildcards (i.e. first match wins).

Values are saved verbatim, except that iteration variables are substituted in them.

Syntax:

config *key value*

Example config lines:

```
config sim-time-limit 90min
config network Aloha
config Aloha.numHosts 10
config Aloha.host[*].iaTime exponential(1s)
config **.x "uniform(0m, 1000m)"
```

J.1.6 Scalar Data

Contains an output scalar value.

Syntax:

scalar *moduleName scalarName value*

Examples:

```
scalar "net.switchA.relay" "processed frames" 100
```

Scalar lines may be immediately followed by *attribute* lines. OMNEST uses the following attributes for scalars:

- **title**: suggested title on charts
- **unit**: measurement unit, e.g. `s` for seconds

J.1.7 Vector Declaration

Defines an output vector.

Syntax:

vector *vectorId moduleName vectorName*

vector *vectorId moduleName vectorName columnSpec*

Where *columnSpec* is a string, encoding the meaning and ordering the columns of data lines. Characters of the string mean:

- **E** event number
- **T** simulation time
- **V** vector value

Common values are `TV` and `ETV`. The default value is `TV`.

Vector lines may be immediately followed by *attribute* lines. OMNEST uses the following attributes for vectors:

- **title**: suggested vector title on charts
- **unit**: measurement unit, e.g. `s` for seconds
- **enum**: symbolic names for values of the vector; syntax is `"IDLE=0, BUSY=1, OFF=2"`
- **type**: data type, one of `int`, `double` and `enum`
- **interpolationmode**: hint for interpolation mode on the chart: `none` (=do not connect the dots), `sample-hold`, `backward-sample-hold`, `linear`
- **min**: minimum value
- **max**: maximum value

J.1.8 Vector Data

Adds a value to an output vector. This is the same as in older output vector files.

Syntax:

vectorId column1 column2 ...

Simulation times and event numbers *within an output vector* are required to be in increasing order.

Performance note: Data lines belonging to the same output vector may be written out in clusters (of size roughly a multiple of the disk's physical block size). Then, since an output vector file is typically not kept in memory during analysis, indexing the start offsets of these clusters allows one to read the file and seek in it more efficiently. This does not require any change or extension to the file format.

J.1.9 Index Header

The first line of the index file stores the size and modification date of the vector file. If the attributes of a vector file differ from the information stored in the index file, then the IDE automatically rebuilds the index file.

Syntax:

file *filesize modificationDate*

J.1.10 Index Data

Stores the location and statistics of blocks in the vector file.

Syntax:

vectorId offset length firstEventNo lastEventNo firstSimtime lastSimtime count min max sum sqrsum

where

- *offset*: the start offset of the block
- *length*: the length of the block
- *firstEventNo*, *lastEventNo*: the event number range of the block (optional)
- *firstSimtime*, *lastSimtime*: the simtime range of the block
- *count, min, max, sum, sqrsum*: collected statistics of the values in the block

J.1.11 Statistics Object

Represents a statistics object.

Syntax:

statistic *moduleName statisticName*

Example:

```
statistic Aloha.server  "collision multiplicity"
```

A *statistic* line may be followed by *field* and *attribute* lines, and a series of *bin* lines that represent histogram data.

OMNEST uses the following attributes:

- **title**: suggested title on charts
- **unit**: measurement unit, e.g. `s` for seconds
- **type**: type of the collected values: `int` or `double`; the default is `double`

A full example with fields, attributes and histogram bins:

```
statistic Aloha.server  "collision multiplicity"
field count 13908
field mean 6.8510209951107
field stddev 5.2385484477843
field sum 95284
field sqrsum 1034434
field min 2
field max 65
attr type int
attr unit packets
bin     -INF    0
bin     0       0
bin     1       0
bin     2       2254
bin     3       2047
bin     4       1586
bin     5       1428
bin     6       1101
bin     7       952
bin     8       785
...
bin     52      2
```

J.1.12 Field

Represents a field in a statistics object.

Syntax:

field *fieldName value*

Example:

```
field sum 95284
```

Fields:

- **count**: observation count
- **mean**: mean of the observations

- **stddev**: standard deviation
- **min**: minimum of the observations
- **max**: maximum of the observations
- **sum**: sum of the observations
- **sqrsum**: sum of the squared observations

For weighted statistics, *sum* and *sqrsum* are replaced by the following fields:

- **weights**: sum of the weights
- **weightedSum**: the weighted sum of the observations
- **sqrSumWeights**: sum of the squared weights
- **weightedSqrSum**: weighted sum of the squared observations

J.1.13 Histogram Bin

Represents a bin in a histogram object.

Syntax:

bin *binLowerBound value*

Histogram name and module is defined on the **statistic** line, which is followed by several **bin** lines to contain data. Any non-**bin** line marks the end of the histogram data.

The *binLowerBound* column of **bin** lines represent the (inclusive) lower bound of the given histogram cell. **Bin** lines are in increasing *binLowerBound* order.

The *value* column of a **bin** line represents the observation count in the given cell: *value k* is the number of observations greater or equal to *binLowerBound k*, but smaller than *binLowerBound k+1*. *Value* is not necessarily an integer, because the cKSplit and cPSquare algorithms produce non-integer estimates. The first **bin** line is the underflow cell, and the last **bin** line is the overflow cell.

Example:

```
bin -INF  0
bin 0 4
bin 2 6
bin 4 2
bin 6 1
```

J.2 SQLite Result Files

The database structure in SQLite result files is created with the following SQL statements. Scalar and vector files are identical in structure, they only differ in data.

```
CREATE TABLE run
(
    runId       INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    runName     TEXT NOT NULL,
    simtimeExp  INTEGER NOT NULL
);

CREATE TABLE runAttr
(
    runId       INTEGER  NOT NULL REFERENCES run(runId) ON DELETE CASCADE,
    attrName    TEXT NOT NULL,
    attrValue   TEXT NOT NULL
);

CREATE TABLE runItervar
(
    runId       INTEGER  NOT NULL REFERENCES run(runId) ON DELETE CASCADE,
    itervarName TEXT NOT NULL,
    itervarValue TEXT NOT NULL
);

CREATE TABLE runConfig
(
    runId       INTEGER NOT NULL REFERENCES run(runId) ON DELETE CASCADE,
    configKey   TEXT NOT NULL,
    configValue TEXT NOT NULL,
    configOrder INTEGER NOT NULL
);

CREATE TABLE scalar
(
    scalarId      INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    runId         INTEGER  NOT NULL REFERENCES run(runId) ON DELETE CASCADE,
    moduleName    TEXT NOT NULL,
    scalarName    TEXT NOT NULL,
    scalarValue   REAL        -- cannot be NOT NULL, because sqlite converts NaN d
);

CREATE TABLE scalarAttr
(
    scalarId      INTEGER  NOT NULL REFERENCES scalar(scalarId) ON DELETE CASCADE,
    attrName      TEXT NOT NULL,
    attrValue     TEXT NOT NULL
);

CREATE TABLE parameter
(
    paramId       INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    runId         INTEGER  NOT NULL REFERENCES run(runId) ON DELETE CASCADE,
    moduleName    TEXT NOT NULL,
    paramName     TEXT NOT NULL,
    paramValue    TEXT NOT NULL
```

```
);

CREATE TABLE paramAttr
(
    paramId       INTEGER  NOT NULL REFERENCES parameter(paramId) ON DELETE CASCAD
    attrName      TEXT NOT NULL,
    attrValue     TEXT NOT NULL
);

CREATE TABLE statistic
(
    statId        INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    runId         INTEGER NOT NULL REFERENCES run(runId) ON DELETE CASCADE,
    moduleName    TEXT NOT NULL,
    statName      TEXT NOT NULL,
    isHistogram   INTEGER NOT NULL,   -- actually, BOOL
    isWeighted    INTEGER NOT NULL,   -- actually, BOOL
    statCount     INTEGER NOT NULL,
    statMean      REAL,   -- note: computed; stored for convenience
    statStddev    REAL,   -- note: computed; stored for convenience
    statSum       REAL,
    statSqrsum    REAL,
    statMin       REAL,
    statMax       REAL,
    statWeights          REAL,  -- note: names of this and subsequent fields are c
    statWeightedSum      REAL,
    statSqrSumWeights    REAL,
    statWeightedSqrSum   REAL
);

CREATE TABLE statisticAttr
(
    statId        INTEGER NOT NULL REFERENCES statistic(statId) ON DELETE CASCADE,
    attrName      TEXT NOT NULL,
    attrValue     TEXT NOT NULL
);

CREATE TABLE histogramBin
(
    statId        INTEGER NOT NULL REFERENCES statistic(statId) ON DELETE CASCADE,
    lowerEdge     REAL NOT NULL,
    binValue      REAL NOT NULL
);

CREATE TABLE vector
(
    vectorId        INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    runId           INTEGER NOT NULL REFERENCES run(runId) ON DELETE CASCADE,
    moduleName      TEXT NOT NULL,
    vectorName      TEXT NOT NULL,
    vectorCount     INTEGER,    -- cannot be NOT NULL because we fill it in later
    vectorMin       REAL,
```

```
    vectorMax       REAL,
    vectorSum       REAL,
    vectorSumSqr    REAL,
    startEventNum   INTEGER,
    endEventNum     INTEGER,
    startSimtimeRaw INTEGER,
    endSimtimeRaw   INTEGER

);

CREATE TABLE vectorAttr
(
    vectorId      INTEGER  NOT NULL REFERENCES vector(vectorId) ON DELETE CASCADE,
    attrName      TEXT NOT NULL,
    attrValue     TEXT NOT NULL
);

CREATE TABLE vectorData
(
    vectorId      INTEGER NOT NULL REFERENCES vector(vectorId) ON DELETE CASCADE,
    eventNumber   INTEGER NOT NULL,
    simtimeRaw    INTEGER NOT NULL,
    value         REAL  -- cannot be NOT NULL because of NaN values
);
```

Notes:

1. To preserve precision, simulation time is stored in raw form, i.e. the underlying `int64` is stored as an integer. To get the real value, they have to be multiplied by 10 to the power of the simtime exponent, which is global for the simulation run. The simtime exponent is stored in the `simtimeExp` column of the `run` table.
2. Some columns like vector statistics are not marked as `NOT NULL`, because of technical reasons: their values are not available at the time of the insertion, only at the end of the simulation.
3. `REAL` columns are not marked as `NOT NULL`, because SQLite stores floating-point NaN values as `NULL`s.

CAUTION: SQLite support in OMNEST is currently experimental, so the above database structure may change in future releases.

Appendix K

Eventlog File Format

This appendix documents the format of the eventlog file. Eventlog files are written by the simulation (when enabled). Everything that happens during the simulation is recorded into the file, [1] so the file can later be used to reproduce the history of the simulation on a sequence chart, or in some other form.

The file is a line-oriented text file. Blank lines and lines beginning with "#" (comments) will be ignored. Other lines begin with an *entry identifier* like `E` for *Event* or `BS` for *BeginSend*, followed by *attribute-identifier* and *value* pairs. One exception is debug output (recorded from `EV«...` statements), which are represented by lines that begin with a hyphen, and continue with the actual text.

The grammar of the eventlog file is the following:

```
<file> ::= <line>*
<line> ::= <empty-line> | <user-log-message> | <event-log-entry>
<empty-line> ::= CR LF
<user-log-message> ::= - SPACE <text> CR LF
<event-log-entry> ::= <event-log-entry-type> SPACE <parameters> CR LF
<event-log-entry-type> ::= SB | SE | BU | MB | ME | MC | MD | MR | GC | GD |
                           CC | CD | CS | MS | CE | BS | ES | SD | SH | DM | E
<parameters> ::= (<parameter>)*
<parameter> ::= <name> SPACE <value>
<name> ::= <text>
<value> ::= <boolean> | <integer> | <text> | <quoted-text>
```

The eventlog file must also fulfill the following requirements:

- simulation events are in increasing event number and simulation time order

Here is a fragment of an existing eventlog file as an example:

```
E # 14 t 1.018454036455 m 8 ce 9 msg 6
BS id 6 tid 6 c cMessage n send/endTx pe 14
ES t 4.840247053855
MS id 8 d t=TRANSMIT,,#808000;i=device/pc_s
MS id 8 d t=,,#808000;i=device/pc_s
```

[1] With certain granularity of course, and subject to filters that were active during simulation

```
E # 15 t 1.025727827674 m 2 ce 13 msg 25
- another frame arrived while receiving -- collision!
CE id 0 pe 12
BS id 0 tid 0 c cMessage n end-reception pe 15
ES t 1.12489449434
BU id 2 txt "Collision! (3 frames)"
DM id 25 pe 15
```

K.1 Supported Entry Types and Their Attributes

The following entries and attributes are supported in the eventlog file:

SB *(SimulationBegin)*: mandatory first line of the eventlog file, followed by an empty line

- **ov** (*omnetppVersion*, int): OMNeT++ version, e.g. 0x0401 (=1025) is release 4.1
- **ev** (*eventlogVersion*, int): eventlog version number
- **rid** (*runId*, string): identifies the simulation run

SE *(SimulationEnd)*: optional last non-empty line of the eventlog file, followed by an empty line

- **e** (*isError*, bool): specifies if the simulation terminated due to an error
- **c** (*resultCode*, int): the error code in case of an error, otherwise the normal result code
- **m** (*message*, string): human readable description

E *(Event)*: an event that is processing a message, terminated by an empty line

- **#** (*eventNumber*, eventnumber_t): unique event number
- **t** (*simulationTime*, simtime_t): simulation time when the event occurred
- **m** (*moduleId*, int): id of the processing module
- **ce** (*causeEventNumber*, eventnumber_t): event number from which the message being processed was sent, or -1 if the message was sent from initialize
- **msg** (*messageId*, msgid_t): id of the message being processed
- **f** (*fingerprints*, string): current simulation fingerprints

S *(Snapshot)*: a snapshot of the current simulation state, followed by state entries, and terminated by an emtpy line

- **f** (*fileOffset*, int64_t): file offset of this entry
- **#** (*eventNumber*, eventnumber_t): event number of the last processed event
- **t** (*simulationTime*, simtime_t): simulation time of the last processed event

I *(Index)*: incremental snapshot specifying additional and removed entries with an event number and a line index, followed by an empty line

- **f** (*fileOffset*, int64_t): file offset of this entry
- **i** (*previousIndexFileOffset*, int64_t): file offset of the previous index entry
- **s** (*previousSnapshotFileOffset*, int64_t): file offset of the previous snapshot entry
- **#** (*eventNumber*, eventnumber_t): event number of the last processed event
- **t** (*simulationTime*, simtime_t): simulation time of the last processed event

abstract *(Reference)*: base class for index entry references

- **#** (*eventNumber*, eventnumber_t): event number of the last referred event
- **b** (*beginEntryIndex*, int): begin index of the referred entry within the event
- **e** (*endEntryIndex*, int): end index of the referred entry within the event

RF *(ReferenceFound)*: specifies an eventlog entry found in the snapshot

- no parameters

RA *(ReferenceAdded)*: specifies an eventlog entry added to the index

- no parameters

RR *(ReferenceRemoved)*: specifies an eventlog entry removed from the index

- no parameters

abstract *(ModuleReference)*: base class for entries referring to a module

- **id** (*moduleId*, int): id of the module

abstract *(GateReference)*: base class for entries referring to a gate

- **m** (*moduleId*, int): id of module where the gate is
- **g** (*gateId*, int): id of the gate

abstract *(ConnectionReference)*: base class for entries referring to a connection

- **sm** (*sourceModuleId*, int): id of the source module identifying the connection
- **sg** (*sourceGateId*, int): id of the gate at the source module identifying the connection

abstract *(MessageReference)*: base class for entries referring to a message

- **id** (*messageId*, msgid_t): id of the message

abstract *(ModuleDescription)*: base class for entries describing a module

- **c** (*moduleClassName*, string): C++ class name of the module
- **t** (*nedTypeName*, string): fully qualified NED type name
- **pid** (*parentModuleId*, int): id of the parent module
- **n** (*fullName*, string): full dotted hierarchical module name
- **cm** (*compoundModule*, bool): whether module is a simple or compound module

abstract *(GateDescription)*: base class for entries describing a gate

- **n** (*name*, string): gate name
- **i** (*index*, int): gate index if vector, -1 otherwise
- **o** (*isOutput*, bool): whether the gate is input or output

abstract *(ConnectionDescription)*: base class for entries describing a connection

- **dm** (*destModuleId*, int): id of the destination module
- **dg** (*destGateId*, int): id of the gate at the destination module

abstract *(MessageDescription)*: base class for entries describing a message

- **tid** (*messageTreeId*, msgid_t): id of the message inherited by dup
- **eid** (*messageEncapsulationId*, msgid_t): id of the message inherited by encapsulation
- **etid** (*messageEncapsulationTreeId*, msgid_t): id of the message inherited by both dup and encapsulation
- **c** (*messageClassName*, string): C++ class name of the message
- **n** (*messageName*, string): message name
- **k** (*messageKind*, short): message kind
- **p** (*messagePriority*, short): message priority
- **l** (*messageLength*, int64_t): message length in bits
- **er** (*hasBitError*, bool): true indicates that the message has bit errors
- **m** (*ownerModuleId*, int): id of the owner module where the message was found or -1 (FES)
- **sm** (*senderModuleId*, int): id of the source module where the message was sent from
- **sg** (*senderGateId*, int): id of the gate at the source module from which the message is being sent
- **st** (*sendingTime*, simtime_t): simulation time when the message was sent
- **am** (*arrivalModuleId*, int): id of the destination module where the message was sent to

- **ag** (*arrivalGateId*, int): id of the gate at the source module from which the message is being sent
- **at** (*arrivalTime*, simtime_t): simulation time when the message will arrive
- **d** (*detail*, string): detailed information of message content when recording message data is turned on
- **pe** (*previousEventNumber*, eventnumber_t): event number from which the message being cloned was sent, or -1 if the message was sent from initialize

abstract *(ModuleDisplayString)*: base class for entries describing a module display string

- **d** (*displayString*, string): the new display string

abstract *(GateDisplayString)*: base class for entries describing a gate display string

- **d** (*displayString*, string): the new display string

abstract *(ConnectionDisplayString)*: base class for entries describing a connection display string

- **d** (*displayString*, string): the new display string

abstract *(MessageDisplayString)*: base class for entries describing a message display string

- **d** (*displayString*, string): the new display string

CMB *(ComponentMethodBegin)*: beginning of a call to another module

- **sm** (*sourceComponentId*, int): id of the caller component
- **tm** (*targetComponentId*, int): id of the component being called
- **m** (*methodName*, string): C++ method name

CME *(ComponentMethodEnd)*: end of a call to another component

- no parameters

MC *(ModuleCreated)*: creating a module

- no parameters

MD *(ModuleDeleted)*: deleting a module

- no parameters

GC *(GateCreated)*: creating a gate

- no parameters

GD *(GateDeleted)*: deleting a gate

- no parameters

CC *(ConnectionCreated)*: creating a connection

- no parameters

CD *(ConnectionDeleted)*: deleting a connection

- no parameters

MDC *(ModuleDisplayStringChanged)*: a module display string change

- no parameters

GDC *(GateDisplayStringChanged)*: a gate display string change

- no parameters

CDC *(ConnectionDisplayStringChanged)*: a connection display string change

- no parameters

EDC *(MessageDisplayStringChanged)*: a message display string change

- no parameters

CM *(CreateMessage)*: creating a message

- no parameters

CL *(CloneMessage)*: cloning a message either via the copy constructor or by dup

- **cid** (*cloneId*, msgid_t): id of the original message that is cloned

DM *(DeleteMessage)*: deleting a message

- no parameters

EN *(EncapsulatePacket)*: encapsulating a packet

- **cid** (*encapsulatedPacketId*, msgid_t): id of the encapsulated packet

DE *(DecapsulatePacket)*: decapsulating a packet

- **cid** (*encapsulatedPacketId*, msgid_t): id of the encapsulated packet

BS *(BeginSend)*: beginning to send a message

- **sd** (*sendDelay*, simtime_t): send after this delay
- **up** (*isUpdate*, bool): whether this is a transmission update
- **tx** (*transmissionId*, txid_t): for pairing transmission updates with the original transmission

ES *(EndSend)*: prediction of the arrival of a message, only a message reference because can't be alone

- **i** (*isDeliveredImmediately*, bool): true indicates the message is delivered

SD *(SendDirect)*: sending a message directly to a destination gate

- **sm** (*senderModuleId*, int): id of the source module from which the message is being sent
- **dm** (*destModuleId*, int): id of the destination module to which the message is being sent
- **dg** (*destGateId*, int): id of the gate at the destination module to which the message is being sent
- **pd** (*propagationDelay*, simtime_t): propagation delay as the message is propagated through the connection
- **td** (*transmissionDelay*, simtime_t): transmission duration as the whole message is sent from the source gate
- **rd** (*remainingDuration*, simtime_t): remaining transmission time (if packet is a tx update)

SH *(SendHop)*: sending a message through a connection identified by its source module and gate id

- **sm** (*senderModuleId*, int): id of the source module from which the message is being sent
- **sg** (*senderGateId*, int): id of the gate at the source module from which the message is being sent
- **pd** (*propagationDelay*, simtime_t): propagation delay as the message is propagated through the connection
- **td** (*transmissionDelay*, simtime_t): transmission duration as the whole message is sent from the source gate
- **rd** (*remainingDuration*, simtime_t): remaining transmission time (if packet is a tx update)
- **d** (*discard*, bool): whether the channel has discarded the message

CE *(CancelEvent)*: canceling an event caused by sending a self message

- no parameters

MF *(ModuleFound)*: a module found in the simulation while traversing the modules (used in snapshots)

- **#** (*lastSeenEventNumber*, eventnumber_t): event number of the module created entry
- **ei** (*lastSeenEntryIndex*, int): index of the module created entry

GF *(GateFound)*: a gate found in the simulation while traversing the modules (used in snapshots)

- **#** (*lastSeenEventNumber*, eventnumber_t): event number of the gate created entry
- **ei** (*lastSeenEntryIndex*, int): index of the gate created entry

CF *(ConnectionFound)*: a connection found in the simulation while traversing the modules (used in snapshots)

- **#** (*lastSeenEventNumber*, eventnumber_t): event number of the connection created entry
- **ei** (*lastSeenEntryIndex*, int): index of the connection created entry

EF *(MessageFound)*: a message found in the future event queue (FES) or while traversing the modules (used in snapshots)

- no parameters

MDF *(ModuleDisplayStringFound)*: a module display string found (used in snapshots)

- **#** (*lastSeenEventNumber*, eventnumber_t): event number of the module display string changed entry
- **ei** (*lastSeenEntryIndex*, int): index of the module display string changed entry

GDF *(GateDisplayStringFound)*: a gate display string found (used in snapshots)

- **#** (*lastSeenEventNumber*, eventnumber_t): event number of the gate display string changed entry
- **ei** (*lastSeenEntryIndex*, int): index of the gate display string changed entry

CDF *(ConnectionDisplayStringFound)*: a connection display string found (used in snapshots)

- **#** (*lastSeenEventNumber*, eventnumber_t): event number of the connection display string changed entry
- **ei** (*lastSeenEntryIndex*, int): index of the connection display string changed entry

EDF *(MessageDisplayStringFound)*: a message display string found (used in snapshots)

- **#** (*lastSeenEventNumber*, eventnumber_t): event number of the message display string changed entry
- **ei** (*lastSeenEntryIndex*, int): index of the message display string changed entry

BU *(Bubble)*: display a bubble message

- **id** (*moduleId*, int): id of the module which printed the bubble message
- **txt** (*text*, string): displayed message text

abstract *(CustomReference)*: custom data reference provided by OMNeT users

- **t** (*type*, string): user specified unique type
- **k** (*key*, long): unique key that links custom entries together

abstract *(CustomDescription)*: custom data provided by OMNeT users

- **c** (*content*, string): user specified content (text, CSV, XML, JSON, etc.)

CUC *(CustomCreated)*: created a custom data object

- no parameters

CUD *(CustomDeleted)*: deleted a custom data object

- no parameters

CUM *(CustomChanged)*: changed a custom data object

- no parameters

CUF *(CustomFound)*: found a custom data object (used in snapshots)

- **#** (*lastSeenEventNumber*, eventnumber_t): event number of the corresponding created entry
- **ei** (*lastSeenEntryIndex*, int): index of the corresponding created begin entry

CU *(Custom)*: custom data provided by OMNeT users

- no parameters

Appendix L

Python API for Chart Scripts

This chapter describes the API of the Python modules available for chart scripts. These modules are available in the Analysis Tool in the IDE, in `opp_charttool`, and may also be used in standalone Python scripts.

Some conventional import aliases appear in code fragments throughout this chapter, such as `np` for NumPy and `pd` for Pandas.

L.1 Modules

L.1.1 Module omnetpp.scave.results

Provides access to simulation results loaded from OMNeT++ result files (.sca, .vec). The results are returned as Pandas `DataFrame`'s of various formats.

The module can be used in several ways, depending on the environment it is run in, and on whether the set of result files to query are specified in a stateful or a stateless way:

1. Inside a chart script in the Analysis Tool in the Simulation IDE. In that mode, the set of result files to take as input are defined on the "Inputs" page of the editor. The `get_results()`, `get_scalars()` and similar methods are invoked with a filter string as first argument to select the appropriate subset of results from the result files. Note that this mode of operation is stateful: The state is set up appropriately by the IDE before the chart script is run.

   A similar thing happens when charts in an analysis (.anf) file are run from within `opp_charttool`: the tool sets up the module state before running the chart script, so that the getter methods invoked with a filter string will return result from the set of result files saved as "inputs" in the anf file.

2. Standalone stateful mode. In order to use `get_results()`, `get_scalars()` and similar methods with a filter string, the module needs to be configured via the `set_inputs()`/`add_inputs(`
   functions to tell it the set of result files to use as input for the queries. (Doing so is analogous to filling in the "Inputs" page in the IDE).

3. Stateless mode. It is possible to load the result files (in whole or a subset of results in them) into memory as a "raw" `DataFrame` using `read_result_files()`, and then use `get_scalars()`, `get_vectors()` and other getter functions with the dataframe as their

first argument to produce `DataFrame`'s of other formats. Note that when going this route, a filter string can be specified to `read_result_files()` but not to the getter methods. However, Pandas already provides several ways for filtering the rows of a dataframe, for example by indexing with logical operators on columns, or using the `df.query()`, `df.pipe()` or `df.apply()` methods.

Filter expressions

The `filter_or_dataframe` parameters in all functions must contain either a filter string, or a "raw" dataframe produced by `read_result_files()`. When it contains a filter string, the function operates on the set of result files configured earlier (see stateful mode above).

Filter strings of all functions have the same syntax. It is always evaluated independently on every loaded result item or metadata entry, and its value determines whether the given item or piece of metadata is included in the returned `DataFrame`.

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>`.

The following properties are available:

- `name`: Name of the result or item.
- `module`: Full path of the result's module.
- `type`: Type of the item. Value is one of: `scalar`, `vector`, `parameter`, `histogram`, `statistics`.
- `isfield`: `true` is the item is a synthetic scalar that represents a field of 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`.

Patterns 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 range a-z
- {32..255} any number (i.e. sequence of digits) in range 32..255 (e.g. 99)
- [32..255] any number in square brackets in range 32..255 (e.g. [99])
- \ takes away the special meaning of the subsequent character

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

Example: `module =~ "**.host*" AND (name =~ "pkSent*" OR name =~ "pkRecvd*")`

The "raw" dataframe format

This dataframe format is a central one, because the content of "raw" dataframes correspond exactly to the content result files, i.e. it is possible to convert between result files and the "raw" dataframe format without data loss. The "raw" dataframe format also corresponds in a one-to-one manner to the "CSV-R" export format of the Simulation IDE and `opp_scavetool`.

The outputs of the `get_results()` and `read_result_files()` functions are in this format, and the dataframes that can be passed as input into certain query functions (`get_scalars()`, `get_vectors()`, `get_runs()`, etc.) are also expected in the same format.

Columns of the `DataFrame`:

- `runID` *(string)*: Identifies the simulation run
- `type` *(string)*: Row type, one of the following: scalar, vector, statistics, histogram, runattr, itervar, param, attr
- `module` *(string)*: Hierarchical name (a.k.a. full path) of the module that recorded the result item
- `name` *(string)*: Name of the result item (scalar, statistic, histogram or vector)
- `attrname` *(string)*: Name of the run attribute or result item attribute (in the latter case, the module and name columns identify the result item the attribute belongs to)
- `attrvalue` *(string)*: Value of run and result item attributes, iteration variables, saved ini param settings (runattr, attr, itervar, param)
- `value` *(double or string)*: Output scalar or parameter value
- `count`, `sumweights`, `mean`, `min`, `max`, `stddev` *(double)*: Fields of the statistics or histogram
- `binedges`, `binvalues` *(np.array)*: Histogram bin edges and bin values. `len(binedges)==len(binva`
- `underflows`, `overflows` *(double)*: Sum of weights (or counts) of underflown and overflown samples of histograms
- `vectime`, `vecvalue` *(np.array)*: Output vector time and value arrays

Requesting metadata columns

Several query functions have the `include_attrs`, `include_runattrs`, `include_itervars`, `include_param_assignments`, and `include_config_entries` boolean options. When such an option is turned on, it will add extra columns into the returned `DataFrame`, one for each result attribute, run attribute, iteration variable, etc. When there is a name clash among items of different types, the column name for the second one will be modified by adding its type after an underscore (`_runattr`, `_itervar`, `_config`, `_param`).

- include_attrs: Adds the attributes of the result in question
- include_runattrs: Adds the run attributes of the (result's) run
- include_itervars: Adds the iteration variables of the (result's) run
- include_config_entries: Adds all configuration entries of the (result's) run, including parameter parameter assignments and per-object configuration options. If this option is turned on, include_param_assignments has no effect.
- include_param_assignments: Adds the configuration entries that set module or channel parameters. This is a subset of the entries added by include_config_entries.

Note that values in metadata columns are generally strings (with missing values represented as None or nan). The Pandas to_numeric() function or utils.to_numeric() can be used to convert values to float or int where needed.

Class ResultQueryError

get_serial()

```
get_serial()
```

Returns an integer that is incremented every time the set of loaded results change, typically as a result of the IDE loading, reloading or unloading a scalar or vector result file. The serial can be used for invalidating cached intermediate results when their input changes.

set_inputs()

```
set_inputs(filenames)
```

Specifies the set of simulation result files (.vec, .sca) to use as input for the query functions. The argument may be a single string, or a list of strings. Each string is interpreted as a file or directory path, and may also contain wildcards. In addition to ? and *, ** (which is able to match several directory levels) is also accepted as wildcard. If a path corresponds to a directory, it is interpreted as ["<dir>/**/*.sca", "<dir>/**/*.vec"], that is, all result files will be loaded from that directory and recursively all its subdirectories.

Examples: set_inputs("results/"), set_inputs("results/**.sca"), set_inputs(["config1/*.s
config2/.sca"]).

add_inputs()

```
add_inputs(filenames)
```

Appends to the set of simulation result files (.vec, .sca) to use as input for the query functions. The argument may be a single string, or a list of strings. Each string is interpreted as a file or directory path, and may also contain wildcards (?, *, **). See set_inputs() for more details.

read_result_files()

```
read_result_files(filenames, filter_expression=None,
include_fields_as_scalars=False, vector_start_time=-inf,
vector_end_time=inf)
```

Loads the simulation result files specified in the first argument `filenames`, and returns the filtered set of results and metadata as a Pandas `DataFrame`.

The `filenames` argument specifies the set of simulation result files (.vec, .sca) to load. The argument may be a single string, or a list of strings. Each string is interpreted as a file or directory path, and may also contain wildcards (`?`, `*`, `**`). See `set_inputs()` for more details on this format.

It is possible to limit the set of results to return by specifying a filter expression, and vector start/end times.

Parameters:

- `filenames` *(string, or list of strings)*: Specifies the result files to load.
- `filter_expression` *(string)*: The filter expression to select the desired items to load. Example: `module =~ "*host*" AND name =~ "numPacket*"`
- `include_fields_as_scalars` *(bool)*: Optional. If `True`, the fields of statistics and histograms (`:min`, `:mean`, etc.) are also returned as synthetic scalars.
- `vector_start_time`, `vector_end_time` *(double)*: Optional time limits to trim the data of vector type results. The unit is seconds, the interval is left-closed, right-open.

Returns: a `DataFrame` in the "raw" format (see the corresponding section of the module documentation for details).

get_results()

```
get_results(filter_or_dataframe="", row_types=None,
omit_unused_columns=True, include_fields_as_scalars=False, start_time=-inf,
end_time=inf)
```

Returns a filtered set of results and metadata in a Pandas `DataFrame`. The items can be any type, even mixed together in a single `DataFrame`. They are selected from the complete set of data referenced by the analysis file (`.anf`), including only those for which the given `filter_or_dataframe` evaluates to `True`.

Parameters:

- `filter_or_dataframe` *(string or dataframe)*: The filter expression to select the desired items from the inputs, or a dataframe in the "raw" format. Example: `module =~ "*host*" AND name =~ "numPacket*"`
- `row_types`: Optional. When given, filters the returned rows by type. Should be a unique list, containing any number of these strings: `"runattr"`, `"itervar"`, `"config"`, `"scalar"`, `"vector"`, `"statistic"`, `"histogram"`, `"param"`, `"attr"`

- `omit_unused_columns` *(bool)*: Optional. If `True`, all columns that would only contain `None` are removed from the returned DataFrame
- `include_fields_as_scalars` *(bool)*: Optional. If `True`, the fields of statistics and histograms (`:min`, `:mean`, etc.) are also returned as synthetic scalars.
- `start_time`, `end_time` *(double)*: Optional time limits to trim the data of vector type results. The unit is seconds, both the `vectime` and `vecvalue` arrays will be affected, the interval is left-closed, right-open.

Returns: a `DataFrame` in the "raw" format (see the corresponding section of the module documentation for details).

get_runs()

```
get_runs(filter_or_dataframe="", include_runattrs=False,
include_itervars=False, include_param_assignments=False,
include_config_entries=False)
```

Returns a filtered list of runs, identified by their run ID.

Parameters:

- `filter_or_dataframe` *(string or dataframe)*: The filter expression to select the desired run from the inputs, or a dataframe in the "raw" format (e.g. one returned by `read_result_files()`). Example: `runattr:network =~ "Aloha" AND config:Aloha.slotTime =~ 0`
- `include_runattrs`, `include_itervars`, `include_param_assignments`, `include_config_entrie` *(bool)*: Optional. When set to `True`, additional pieces of metadata about the run is appended to the result, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- `runID` *(string)*: Identifies the simulation run
- Additional metadata items (run attributes, iteration variables, etc.), as requested

get_runattrs()

```
get_runattrs(filter_or_dataframe="", include_runattrs=False,
include_itervars=False, include_param_assignments=False,
include_config_entries=False)
```

Returns a filtered list of run attributes.

The set of run attributes is fixed: `configname`, `datetime`, `experiment`, `inifile`, `iterationvars`, `iterationvarsf`, `measurement`, `network`, `processid`, `repetition`, `replication`, `resultdir`, `runnumber`, `seedset`.

Parameters:

- `filter_or_dataframe` *(string or dataframe)*: The filter expression to select the desired run attributes from the inputs, or a dataframe in the "raw" format. Example: `name =~ *date* AND config:Aloha.slotTime =~ 0`
- `include_runattrs`, `include_itervars`, `include_param_assignments`, `include_config_entrie` *(bool)*: Optional. When set to `True`, additional pieces of metadata about the run is appended to the result, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- `runID` *(string)*: Identifies the simulation run
- `name` *(string)*: The name of the run attribute
- `value` *(string)*: The value of the run attribute
- Additional metadata items (run attributes, iteration variables, etc.)

get_itervars()

```
get_itervars(filter_or_dataframe="", include_runattrs=False,
include_itervars=False, include_param_assignments=False,
include_config_entries=False)
```

Returns a filtered list of iteration variables.

Parameters:

- `filter_or_dataframe` *(string or dataframe)*: The filter expression to select the desired iteration variables from the inputs, or a dataframe in the "raw" format. Example: `name =~ iaMean AND config:Aloha.slotTime =~ 0`
- `include_runattrs`, `include_itervars`, `include_param_assignments`, `include_config_entrie` *(bool)*: Optional. When set to `True`, additional pieces of metadata about the run is appended to the result, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- `runID` *(string)*: Identifies the simulation run
- `name` *(string)*: The name of the iteration variable
- `value` *(string)*: The value of the iteration variable.
- Additional metadata items (run attributes, iteration variables, etc.), as requested

get_scalars()

```
get_scalars(filter_or_dataframe="", include_attrs=False,
include_fields=False, include_runattrs=False, include_itervars=False,
include_param_assignments=False, include_config_entries=False)
```

Returns a filtered list of scalar results.

Parameters:

- `filter_or_dataframe` *(string)*: The filter expression to select the desired scalars, or a dataframe in the "raw" format. Example: `name =~ "channelUtilization*" AND runattr:replication =~ "#0"`
- `include_attrs` *(bool)*: Optional. When set to `True`, result attributes (like `unit` or `source` for example) are appended to the DataFrame, pivoted into columns.
- `include_fields` *(bool)*: Optional. If `True`, the fields of statistics and histograms (`:min`, `:mean`, etc.) are also returned as synthetic scalars.
- `include_runattrs`, `include_itervars`, `include_param_assignments`, `include_config_entrie` *(bool)*: Optional. When set to `True`, additional pieces of metadata about the run is appended to the DataFrame, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- `runID` *(string)*: Identifies the simulation run
- `module` *(string)*: Hierarchical name (a.k.a. full path) of the module that recorded the result item
- `name` *(string)*: The name of the scalar
- `value` *(double)*: The value of the scalar
- Additional metadata items (result attributes, run attributes, iteration variables, etc.), as requested

get_parameters()

```
get_parameters(filter_or_dataframe="", include_attrs=False,
include_runattrs=False, include_itervars=False,
include_param_assignments=False, include_config_entries=False)
```

Returns a filtered list of parameters - actually computed values of individual `cPar` instances in the fully built network.

Parameters are considered "pseudo-results", similar to scalars - except their values are strings. Even though they act mostly as input to the actual simulation run, the actually assigned value of individual `cPar` instances is valuable information, as it is the result of the network setup process. For example, even if a parameter is set up as an expression like `normal(3, 0.4)` from `omnetpp.ini`, the returned DataFrame will contain the single concrete value picked for every instance of the parameter.

Parameters:

- `filter_or_dataframe` *(string)*: The filter expression to select the desired parameters, or a dataframe in the "raw" format. Example: `name =~ "x" AND module =~ Aloha.server`
- `include_attrs` *(bool)*: Optional. When set to `True`, result attributes (like `unit` for example) are appended to the DataFrame, pivoted into columns.

- include_runattrs, include_itervars, include_param_assignments, include_config_entrie *(bool)*: Optional. When set to True, additional pieces of metadata about the run is appended to the DataFrame, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- runID *(string)*: Identifies the simulation run
- module *(string)*: Hierarchical name (a.k.a. full path) of the module that recorded the result item
- name *(string)*: The name of the parameter
- value *(string)*: The value of the parameter.
- Additional metadata items (result attributes, run attributes, iteration variables, etc.), as requested

get_vectors()

```
get_vectors(filter_or_dataframe="", include_attrs=False,
include_runattrs=False, include_itervars=False,
include_param_assignments=False, include_config_entries=False,
start_time=-inf, end_time=inf)
```

Returns a filtered list of vector results.

Parameters:

- filter_or_dataframe *(string)*: The filter expression to select the desired vectors, or a dataframe in the "raw" format. Example: name =~ "radioState*" AND runattr:replication =~ "#0"
- include_attrs *(bool)*: Optional. When set to True, result attributes (like unit or source for example) are appended to the DataFrame, pivoted into columns.
- include_runattrs, include_itervars, include_param_assignments, include_config_entrie *(bool)*: Optional. When set to True, additional pieces of metadata about the run is appended to the DataFrame, pivoted into columns. See the "Metadata columns" section of the module documentation for details.
- start_time, end_time *(double)*: Optional time limits to trim the data of vector type results. The unit is seconds, both the vectime and vecvalue arrays will be affected, the interval is left-closed, right-open.

Columns of the returned DataFrame:

- runID *(string)*: Identifies the simulation run
- module *(string)*: Hierarchical name (a.k.a. full path) of the module that recorded the result item
- name *(string)*: The name of the vector

- `vectime`, `vecvalue` *(np.array)*: The simulation times and the corresponding values in the vector
- Additional metadata items (result attributes, run attributes, iteration variables, etc.), as requested

get_statistics()

```
get_statistics(filter_or_dataframe="", include_attrs=False,
include_runattrs=False, include_itervars=False,
include_param_assignments=False, include_config_entries=False)
```

Returns a filtered list of statistics results.

Parameters:

- `filter_or_dataframe` *(string)*: The filter expression to select the desired statistics, or a dataframe in the "raw" format. Example: `name =~ "collisionLength:stat" AND itervar:iaMean =~ "5"`
- `include_attrs` *(bool)*: Optional. When set to `True`, result attributes (like `unit` or `source` for example) are appended to the DataFrame, pivoted into columns.
- `include_runattrs`, `include_itervars`, `include_param_assignments`, `include_config_entrie` *(bool)*: Optional. When set to `True`, additional pieces of metadata about the run is appended to the DataFrame, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- `runID` *(string)*: Identifies the simulation run
- `module` *(string)*: Hierarchical name (a.k.a. full path) of the module that recorded the result item
- `name` *(string)*: The name of the vector
- `count`, `sumweights`, `mean`, `stddev`, `min`, `max` *(double)*: The characteristic mathematical properties of the statistics result
- Additional metadata items (result attributes, run attributes, iteration variables, etc.), as requested

get_histograms()

```
get_histograms(filter_or_dataframe="", include_attrs=False,
include_runattrs=False, include_itervars=False,
include_param_assignments=False, include_config_entries=False)
```

Returns a filtered list of histogram results.

Parameters:

- `filter_or_dataframe` *(string)*: The filter expression to select the desired histograms, or a dataframe in the "raw" format. Example: `name =~ "collisionMultiplicity:histogram" AND itervar:iaMean =~ "2"`
- `include_attrs` *(bool)*: Optional. When set to `True`, result attributes (like `unit` or `source` for example) are appended to the DataFrame, pivoted into columns.
- `include_runattrs`, `include_itervars`, `include_param_assignments`, `include_config_entrie` *(bool)*: Optional. When set to `True`, additional pieces of metadata about the run is appended to the DataFrame, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- `runID` *(string)*: Identifies the simulation run
- `module` *(string)*: Hierarchical name (a.k.a. full path) of the module that recorded the result item
- `name` *(string)*: The name of the vector
- `count`, `sumweights`, `mean`, `stddev`, `min`, `max` *(double)*: The characteristic mathematical properties of the histogram
- `binedges`, `binvalues` *(np.array)*: The histogram edge locations and the weighted sum of the collected samples in each bin. `len(binedges) == len(binvalues) + 1`
- `underflows`, `overflows` *(double)*: The weighted sum of the samples that fell outside of the histogram bin range in the two directions
- Additional metadata items (result attributes, run attributes, iteration variables, etc.), as requested

get_config_entries()

```
get_config_entries(filter_or_dataframe, include_runattrs=False,
include_itervars=False, include_param_assignments=False,
include_config_entries=False)
```

Returns a filtered list of config entries. That is: parameter assignment patterns; and global and per-object config options.

Parameters:

- `filter_or_dataframe` *(string)*: The filter expression to select the desired config entries, or a dataframe in the "raw" format. Example: `name =~ sim-time-limit AND itervar:numHosts =~ 10`
- `include_runattrs`, `include_itervars`, `include_param_assignments`, `include_config_entrie` *(bool)*: Optional. When set to `True`, additional pieces of metadata about the run is appended to the result, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- `runID` *(string)*: Identifies the simulation run
- `name` *(string)*: The name of the config entry
- `value` *(string)*: The value of the config entry
- Additional metadata items (run attributes, iteration variables, etc.), as requested

get_param_assignments()

```
get_param_assignments(filter_or_dataframe, include_runattrs=False,
include_itervars=False, include_param_assignments=False,
include_config_entries=False)
```

Returns a filtered list of parameter assignment patterns. The result is a subset of what `get_config_entries` would return with the same arguments.

Parameters:

- `filter_or_dataframe` *(string)*: The filter expression to select the desired parameter assignments, or a dataframe in the "raw" format. Example: `name =~ **.flowID AND itervar:numHosts =~ 10`
- `include_runattrs`, `include_itervars`, `include_param_assignments`, `include_config_entrie` *(bool)*: Optional. When set to `True`, additional pieces of metadata about the run is appended to the result, pivoted into columns. See the "Metadata columns" section of the module documentation for details.

Columns of the returned DataFrame:

- `runID` *(string)*: Identifies the simulation run
- `name` *(string)*: The parameter assignment pattern
- `value` *(string)*: The assigned value
- Additional metadata items (run attributes, iteration variables, etc.), as requested

L.1.2 Module omnetpp.scave.chart

Provides access to the properties of the current chart for the chart script.

Note that this module is stateful. It is set up appropriately by the OMNeT++ IDE or `opp_charttool` before the chart script is run.

Class ChartScriptError

Raised by chart scripts when they encounter an error. A message parameter can be passed to the constructor, which will be displayed on the plot area in the IDE.

get_properties()

```
get_properties()
```

Returns the currently set properties of the chart as a `dict` whose keys and values are both all strings.

get_property()

```
get_property(key)
```

Returns the value of a single property of the chart, or `None` if there is no property with the given name (key) set on the chart.

get_name()

```
get_name()
```

Returns the name of the chart as a string.

get_chart_type()

```
get_chart_type()
```

Returns the chart type, one of the strings "bar"/"histogram"/"line"/"matplotlib"

is_native_chart()

```
is_native_chart()
```

Returns True if this chart uses the IDE's built-in plotting widgets.

set_suggested_chart_name()

```
set_suggested_chart_name(name)
```

Sets a proposed name for the chart. The IDE may offer this name to the user when saving the chart.

set_observed_column_names()

```
set_observed_column_names(column_names)
```

Sets the DataFrame column names observed during the chart script. The IDE may use it for content assist when the user edits the legend format string.

L.1.3 Module omnetpp.scave.ideplot

This 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`: most functions and the parameters they accept are a subset of `pyplot`'s interface. If one restricts themselves to a subset of Matplotlib's functionality, switching between `omnetpp.scave.ideplot` and `matplotlib.pyplot` in a chart script may be as simple as much as editing the `import` statement.

When the API is used outside the context of a native plotting widget (such as during the run of `opp_charttool`, or in IDE during image export), the functions are emulated with Matplotlib.

Note that this module is stateful. It is set up appropriately by the OMNeT++ IDE or `opp_charttool` before the chart script is run.

is_native_plot()

```
is_native_plot()
```

Returns True if the script is running in the context of a native plotting widget, and False otherwise.

plot()

```
plot(xs, ys, key=None, label=None, drawstyle=None, linestyle=None,
linewidth=None, color=None, marker=None, markersize=None)
```

Plot y versus x as lines and/or markers. Call `plot` multiple times to plot multiple sets of data.

Parameters:

- `x`, `y` *(array-like or scalar)*: The horizontal / vertical coordinates of the data points.
- `key` *(string)*: Identifies the series in the native plot widget.
- `label` *(string)*: Series label for the legend
- `drawstyle` *(string)*: Matplotlib draw style ('default', 'steps', 'steps-pre', 'steps-mid', 'steps-post')
- `linestyle` *(string)*: Matplotlib line style ('-', '–', '-.', ':', etc}
- `linewidth` *(float)*: Line width in pixels
- `color` *(string)*: Matplotlib color name or abbreviation ('b' for blue, 'g' for green, etc.)
- `marker` *(string)*: Matplotlib marker name ('.', ',', 'o', 'x', '+', etc.)
- `markersize` *(float)*: Size of markers in pixels.

hist()

```
hist(x, bins, key=None, density=False, weights=None, cumulative=False,
bottom=None, histtype="stepfilled", color=None, label=None, linewidth=None,
underflows=0.0, overflows=0.0, minvalue=nan, maxvalue=nan)
```

Make a histogram plot. This function adds one histogram the bar plot; make multiple calls to add multiple histograms.

Parameters:

- `x` *(array-like)*: Input values.
- `bins` *(array-like)*: Bin edges, including left edge of first bin and right edge of last bin.
- `key` *(string)*: Identifies the series in the native plot widget.
- `density` *(bool)*: See `mpl.hist()`.
- `weights` *(array-like)*: Weights.
- `cumulative` *(bool)*: See `mpl.hist()`.
- `bottom` *(float)*: Location of the bottom baseline for bins.
- `histtype` *(string)*: Whether to fill the area under the plot. Accepted values are 'step' and 'stepfilled'.
- `color` *(string)*: Matplotlib color name or abbreviation ('b' for blue, 'g' for green, etc.)
- `label` *(string)*: Series label for the legend
- `linewidth` *(float)*: Line width in pixels
- `underflows`, `overflows`: Number of values / sum of weights outside the histogram bins in both directions.
- `minvalue`, `maxvalue`: The minimum and maximum value, or `nan` if unknown.

Restrictions:

1. Overflow bin data (minvalue, maxvalue, underflows and overflows) is not accepted by `pyplot.hist()`.
2. The native plot widget only accepts a precomputed histogram (using the trick documented for `pyplot.hist()`)

bar()

```
bar(x, height, width=0.8, key=None, label=None, color=None, edgecolor=None)
```

Make a bar plot. This function adds one series to the bar plot; make multiple calls to add multiple series.

The bars are positioned at x with the given alignment. Their dimensions are given by width and height. The vertical baseline is bottom (default 0).

Each of x, height, width, and bottom may either be a scalar applying to all bars, or it may be a sequence of length N providing a separate value for each bar.

Parameters:

- `x` *(sequence of scalars)*: The x coordinates of the bars.
- `height` *(scalar or sequence of scalars)*: The height(s) of the bars.
- `width` *(scalar or array-like)*: The width(s) of the bars.
- `key` *(string)*: Identifies the series in the native plot widget.
- `label` *(string)*: The label of the series the bars represent .
- `color` *(string)*: The fill color of the bars.
- `edgecolor` *(string)*: The edge color of the bars.

The native plot implementation has the following restrictions:

- widths are automatic (parameter is ignored)
- x coordinates are automatic (values are ignored)
- height must be a sequence (cannot be a scalar)
- in multiple calls to bar(), the lengths of the height sequence must be equal (i.e. all series must have the same number of values)
- default color is grey (Matplotlib assigns a different color to each series)

set_property()

```
set_property(key, value)
```

Sets one property of the native plot widget to the given value. When invoked outside the contex of a native plot widget, the function does nothing.

Parameters:

- `key` *(string)*: Name of the property.
- `value` *(string)*: The value to set. If any other type than string is passed in, it will be converted to string.

set_properties()

```
set_properties(props)
```

Sets several properties of the native plot widget. It is functionally equivalent to repeatedly calling `set_property` with the entries of the `props` dictionary. When invoked outside the contex of a native plot widget (TODO?), the function does nothing.

Parameters:

- `props` *(dict)*: The properties to set.

get_supported_property_keys()

```
get_supported_property_keys()
```

Returns the list of property names that the native plot widget supports, such as 'Plot.Title', 'X.Axis.Max' and 'Legend.Display', among many others.

Note: This method has no equivalent in `pyplot`. When the script runs outside the IDE (TODO?), the method returns an empty list.

set_warning()

```
set_warning(warning:  str)
```

Displays the given warning text in the plot.

title()

```
title(label:  str)
```

Sets the plot title to the given string.

xlabel()

```
xlabel(xlabel:  str)
```

Sets the label of the X axis to the given string.

ylabel()

```
ylabel(ylabel:  str)
```

Sets the label of the Y axis to the given string..

xlim()

```
xlim(left=None, right=None)
```

Sets the limits of the X axis.

Parameters:

- `left` *(float)*: The left xlim in data coordinates. Passing None leaves the limit unchanged.
- `right` *(float)*: The right xlim in data coordinates. Passing None leaves the limit unchanged.

ylim()

```
ylim(bottom=None, top=None)
```

Sets the limits of the Y axis.

Parameters:

- `bottom` *(float)*: The bottom ylim in data coordinates. Passing None leaves the limit unchanged.
- `top` *(float)*: The top ylim in data coordinates. Passing None leaves the limit unchanged.

xscale()

```
xscale(value:  str)
```

Sets the scale of the X axis. Possible values are 'linear' and 'log'.

yscale()

```
yscale(value:  str)
```

Sets the scale of the Y axis.

xticks()

```
xticks(ticks=None, labels=None, rotation=0)
```

Sets the current tick locations and labels of the x-axis.

Parameters:

- `ticks` *(array_like)*: A list of positions at which ticks should be placed. You can pass an empty list to disable xticks.
- `labels` *(array_like)*: A list of explicit labels to place at the given locs.
- `rotation` *(float)*: Label rotation in degrees.

grid()

```
grid(b=True, which="major")
```

Configure the grid lines.

Parameters:

- `b` *(bool or `None`)*: Whether to show the grid lines.
- `which` *('major', 'minor' or 'both')*: The grid lines to apply the changes on.

legend()

```
legend(show=None, frameon=None, loc=None)
```

Place a legend on the axes.

Parameters:

- `show` *(bool or `None`)*: Whether to show the legend. TODO does pyplot have this?
- `frameon` *(bool or `None`)*: Control whether the legend should be drawn on a patch (frame). Default is `None`, which will take the value from the resource file.
- `loc` *(string or `None`)*: The location of the legend. Possible values are 'best', 'upper right', 'upper left', 'lower left', 'lower right', 'right', 'center left', 'center right', 'lower center', 'upper center', 'center' (these are the values supported by Matplotlib), plus additionally 'outside top left', 'outside top center', 'outside top right', 'outside bottom left', 'outside bottom center', 'outside bottom right', 'outside left top', 'outside left center', 'outside left bottom', 'outside right top', 'outside right center', 'outside right bottom'.

L.1.4 Module omnetpp.scave.utils

A collection of utility function for data manipulation and plotting, built on top of Pandas data frames and the `chart` and `ideplot` packages from `omnetpp.scave`. Functions in this module have been written largely to the needs of the chart templates that ship with the IDE.

There are some functions which are (almost) mandatory elements in a chart script. These are the following.

If you want style settings in the chart dialog to take effect:

- `preconfigure_plot()`
- `postconfigure_plot()`

If you want image/data export to work:

- `export_image_if_needed()`
- `export_data_if_needed()`

make_legend_label()

```
make_legend_label(legend_cols, row, props={})
```

Produces a reasonably good label text (to be used in a chart legend) for a result row from a DataFrame. The legend label is produced as follows.

First, a base version of the legend label is produced:

1. If the DataFrame contains a `legend` column, its content is used

2. Otherwise, if there is a `legend_format` property, it is used as a format string for producing the legend label. The format string may contain references to other columns of the DataFrame in the "$name" or "${name}" form.
3. Otherwise, the legend label is concatenated from the columns listed in `legend_cols`, a list whose contents is usually produced using the `extract_label_columns()` function.

Second, if there is a `legend_replacements` property, the series of regular expression find-/replace operations described in it are performed. `legend_replacements` is expected to be a multi-line string, where each line contains a replacement in the customary "/findstring/replacement/" form. "findstring" should be a valid regex, and "replacement" is a string that may contain match group references ("\1", "\2", etc.). If "/" is unsuitable as separator, other characters may also be used; common choices are "|" and "!". Similar to the format string (`legend_format`), both "findstring" and "replacement" may contain column references in the "$name" or "${name}" form. (Note that "findstring" may still end in "$" to match the end of the string, it won't collide with column references.)

Possible errors:

- References to nonexistent columns in `legend_format` or `legend_replacements` (`KeyError`)
- Malformed regex in the "findstring" parts of `legend_replacements` (`re.error`)
- Invalid group reference in the "replacement" parts of `legend_replacements` (`re.error`)

Parameters:

- `row` *(named tuple)*: The row from the dataframe.
- `props` *(dict)*: The properties that control how the legend is produced
- `legend_cols` *(list of strings)*: The names of columns chosen for the legend.

plot_bars()

```
plot_bars(df, errors_df=None, meta_df=None, props={})
```

Creates a bar plot from the dataframe, with styling and additional input coming from the properties. Each row in the dataframe defines a series.

Group names (displayed on the x axis) are taken from the column index.

The name of the variable represented by the values can be passed in as the `variable_name` argument (as it is not present in the dataframe); if so, it will become the y axis label.

Error bars can be drawn by providing an extra dataframe of identical dimensions as the main one. Error bars will protrude by the values in the errors dataframe both up and down (i.e. range is 2x error).

To make the legend labels customizable, an extra dataframe can be provided, which contains any columns of metadata for each series.

Colors are assigned automatically. The `cycle_seed` property allows you to select other combinations if the default one is not suitable.

Parameters:

- `df`: the dataframe
- `props` *(dict)*: the properties
- `variable_name` *(string)*: The name of the variable represented by the values.
- `errors_df`: dataframe with the errors (in y axis units)
- `meta_df`: dataframe with the metadata about each series

Notable properties that affect the plot:

- `baseline`: The y value at which the x axis is drawn.
- `bar_placement`: Selects the arrangement of bars: aligned, overlap, stacked, etc.
- `xlabel_rotation`: Amount of counter-clockwise rotation of x axis labels a.k.a. group names, in degrees.
- `title`: Plot title (autocomputed if missing).
- `cycle_seed`: Alters the sequence in which colors are assigned to series.

plot_vectors()

```
plot_vectors(df, props, legend_func=make_legend_label)
```

Creates a line plot from the dataframe, with styling and additional input coming from the properties. Each row in the dataframe defines a series.

Colors and markers are assigned automatically. The `cycle_seed` property allows you to select other combinations if the default one is not suitable.

A function to produce the legend labels can be passed in. By default, `make_legend_label()` is used, which offers many ways to influence the legend via dataframe columns and chart properties. In the absence of more specified settings, the legend is normally computed from columns which best differentiate among the vectors.

Parameters:

- `df`: the dataframe
- `props` *(dict)*: the properties
- `legend_func`: the function to produce custom legend labels

Columns of the dataframe:

- `vectime`, `vecvalue` *(Numpy `ndarray`'s of matching sizes)*: the x and y coordinates for the plot
- `interpolationmode` *(str, optional)*: this column normally comes from a result attribute, and determines how the points will be connected
- `legend` *(optional)*: legend label for the series; if missing, legend labels are derived from other columns

- `name`, `title`, `module`, etc. *(optional)*: provide input for the legend

Notable properties that affect the plot:

- `title`: plot title (autocomputed if missing)
- `drawstyle`: Matplotlib draw style; if present, it overrides the draw style derived from `interpolationmode`.
- `linestyle`, `linecolor`, `linewidth`, `marker`, `markersize`: styling
- `cycle_seed`: Alters the sequence in which colors and markers are assigned to series.

plot_vectors_separate()

```
plot_vectors_separate(df, props, legend_func=make_legend_label)
```

This is very similar to `plot_vectors`, with identical usage. The only difference is in the end result, where each vector will be plotted in its own separate set of axes (coordinate system), arranged vertically, with a shared X axis during navigation.

plot_histograms()

```
plot_histograms(df, props, legend_func=make_legend_label)
```

Creates a histogram plot from the dataframe, with styling and additional input coming from the properties. Each row in the dataframe defines a histogram.

Colors are assigned automatically. The `cycle_seed` property allows you to select other combinations if the default one is not suitable.

A function to produce the legend labels can be passed in. By default, `make_legend_label()` is used, which offers many ways to influence the legend via dataframe columns and chart properties. In the absence of more specified settings, the legend is normally computed from columns which best differentiate among the histograms.

Parameters:

- `df`: The dataframe.
- `props` *(dict)*: The properties.
- `legend_func` *(function)*: The function to produce custom legend labels. See `utils.make_legend_la` for prototype and semantics.

Columns of the dataframe:

- `binedges`, `binvalues` *(array-like, `len(binedges)==len(binvalues)+1`)*: The bin edges and the bin values (count or sum of weights) for the histogram.
- `min`, `max`, `underflows`, `overflows` *(float, optional)*: The minimum/maximum values, and the bin values for the underflow/overflow bins. These four columns must either be all present or all absent from the dataframe.

- `legend` *(string, optional)*: Legend label for the series. If missing, legend labels are derived from other columns.
- `name`, `title`, `module`, etc. *(optional)*: Provide input for the legend.

Notable properties that affect the plot:

- `normalize` *(bool)*: If true, normalize the sum of the bin values to 1. If `normalize` is true (and `cumulative` is false), the probability density function (PDF) will be displayed.
- `cumulative` *(bool)*: If true, show each bin as the sum of the previous bin values plus itself. If both `normalize` and `cumulative` are true, that results in the cumulative density function (CDF) being displayed.
- `show_overflows` *(bool)*: If true, show the underflow/overflow bins.
- `title`: Plot title (autocomputed if missing).
- `drawstyle`: Selects whether to fill the area below the histogram line.
- `linestyle`, `linecolor`, `linewidth`: Styling.
- `cycle_seed`: Alters the sequence in which colors and markers are assigned to series.

plot_lines()

```
plot_lines(df, props, legend_func=make_legend_label)
```

Creates a line plot from the dataframe, with styling and additional input coming from the properties. Each row in the dataframe defines a line.

Colors are assigned automatically. The `cycle_seed` property allows you to select other combinations if the default one is not suitable.

A function to produce the legend labels can be passed in. By default, `make_legend_label()` is used, which offers many ways to influence the legend via dataframe columns and chart properties. In the absence of more specified settings, the legend is normally computed from columns which best differentiate among the lines.

Parameters:

- `df`: The dataframe.
- `props` *(dict)*: The properties.
- `legend_func` *(function)*: The function to produce custom legend labels. See `utils.make_legend_la` for prototype and semantics.

Columns of the dataframe:

- `x`, `y` *(array-like, `len(x)==len(y)`)*: The X and Y coordinates of the points.
- `error` *(array-like, `len(x)==len(y)`, optional)*: The half lengths of the error bars for each point.

- `legend` *(string, optional)*: Legend label for the series. If missing, legend labels are derived from other columns.
- `name`, `title`, `module`, etc. *(optional)*: Provide input for the legend.

Notable properties that affect the plot:

- `title`: Plot title (autocomputed if missing).
- `linewidth`: Line width.
- `marker`: Marker style.
- `linestyle`, `linecolor`, `linewidth`: Styling.
- `error_style`: If `error` is present, controls how the error is shown. Accepted values: "Error bars", "Error band"
- `cycle_seed`: Alters the sequence in which colors and markers are assigned to series.

plot_boxwhiskers()

```
plot_boxwhiskers(df, props, legend_func=make_legend_label)
```

Creates a box and whiskers plot from the dataframe, with styling and additional input coming from the properties. Each row in the dataframe defines one set of a box and two whiskers.

Colors are assigned automatically. The `cycle_seed` property allows you to select other combinations if the default one is not suitable.

A function to produce the legend labels can be passed in. By default, `make_legend_label()` is used, which offers many ways to influence the legend via dataframe columns and chart properties. In the absence of more specified settings, the legend is normally computed from columns which best differentiate among the boxes.

Parameters:

- `df`: The dataframe.
- `props` *(dict)*: The properties.
- `legend_func` *(function)*: The function to produce custom legend labels. See `utils.make_legend_la` for prototype and semantics.

Columns of the dataframe:

- `min`, `max`, `mean`, `stddev`, `count` *(float)*: The minimum/maximum values, mean, standard deviation, and sample count of the data.
- `legend` *(string, optional)*: Legend label for the series. If missing, legend labels are derived from other columns.
- `name`, `title`, `module`, etc. *(optional)*: Provide input for the legend.

Notable properties that affect the plot:

- `title`: Plot title (autocomputed if missing).
- `cycle_seed`: Alters the sequence in which colors and markers are assigned to series.

customized_box_plot()

```
customized_box_plot(percentiles, labels=None, axes=None, redraw=True,
*args, **kwargs)
```

Generates a customized box-and-whiskers plot based on explicitly specified percentile values. This method is necessary because `pyplot.boxplot()` insists on computing the stats from the raw data (which we often don't have) itself.

The data are in the `percentiles` argument, which should be list of tuples. One box will be drawn for each tuple. Each tuple contains 6 elements (or 5, because the last one is optional):

(*q1_start*, *q2_start*, *q3_start*, *q4_start*, *q4_end*, *fliers*)

The first five elements have following meaning:

- *q1_start*: y coord of bottom whisker cap
- *q2_start*: y coord of bottom of the box
- *q3_start*: y coord of median mark
- *q4_start*: y coord of top of the box
- *q4_end*: y coord of top whisker cap

The last element, *fliers*, is a list, containing the values of the outlier points.

x coords of the box-and-whiskers plots are automatic.

Parameters:

- `percentiles`: The list of tuples.
- `labels`: If provided, the legend labels for the boxes.
- `axes`: The axes object of the plot.
- `redraw`: If False, redraw is deferred.
- `args`, `kwargs`: Passed to `axes.boxplot()`.

preconfigure_plot()

```
preconfigure_plot(props)
```

Configures the plot according to the given properties, which normally get their values from setting in the "Configure Chart" dialog. Calling this function before plotting was performed should be a standard part of chart scripts.

A partial list of properties taken into account for native plots:

- property keys understood by the plot widget, see `ideplot.get_supported_property_keys()`

And for Matplotlib plots:

- `plt.style`

- properties listed in the `matplotlibrc` property
- properties prefixed with `matplotlibrc.`

Parameters:

- `props` *(dict)*: the properties

postconfigure_plot()

```
postconfigure_plot(props)
```

Configures the plot according to the given properties, which normally get their values from setting in the "Configure Chart" dialog. Calling this function after plotting was performed should be a standard part of chart scripts.

A partial list of properties taken into account:

- `yaxis_title`, `yaxis_title`, `xaxis_min`, `xaxis_max`, `yaxis_min`, `yaxis_max`, `xaxis_log`, `yaxis_log`, `legend_show`, `legend_border`, `legend_placement`, `grid_show`, `grid_density`
- properties listed in the `plot.properties` property

Parameters:

- `props` *(dict)*: the properties

export_image_if_needed()

```
export_image_if_needed(props)
```

If a certain property is set, save the plot in the selected image format. Calling this function should be a standard part of chart scripts, as it is what makes the "Export image" functionality of the IDE and `opp_charttool` work.

Note that for export, even IDE-native charts are rendered using Matplotlib.

The properties that are taken into account:

- `export_image` *(boolean)*: Controls whether to perform the exporting. This is normally `false`, and only set to `true` by the IDE or opp_charttool when image export is requested.
- `image_export_format`: The default is SVG. Accepted formats (and their names) are the ones supported by Matplotlib.
- `image_export_folder`: The folder in which the image file is to be created.
- `image_export_filename`: The output file name. If it has no extension, one will be added based on the format. If missing or empty, a sanitized version of the chart name is used.
- `image_export_width`: Image width in inches (default: 6")
- `image_export_height`: Image height in inches (default: 4")

- image_export_dpi: DPI setting, default 96. For raster image formats, the image dimensions are produced as width (or height) times dpi.

Note that these properties come from two sources to allow meaningful batch export. export_image, image_export_format, image_export_folder and image_export_dpi come from the export dialog because they are common to all charts, while image_export_filename, image_export_width and image_export_height come from the chart properties because they are specific to each chart. Note that image_export_dpi is used for controlling the resolution (for raster image formats) while letting charts maintain their own aspect ratio and relative sizes.

Parameters:

- props *(dict)*: the properties

get_image_export_filepath()

```
get_image_export_filepath(props)
```

Returns the file path for the image to export based on the image_export_format, image_export_folder and image_export_filename properties given in props. If a relative filename is returned, it is relative to the working directory when the image export takes place.

export_data_if_needed()

```
export_data_if_needed(df, props, **kwargs)
```

If a certain property is set, save the dataframe in CSV format. Calling this function should be a standard part of chart scripts, as it is what makes the "Export data" functionality of the IDE and opp_charttool work.

The properties that are taken into account:

- export_data *(boolean)*: Controls whether to perform the exporting. This is normally false, and only set to true by the IDE or opp_charttool when data export is requested.
- data_export_folder: The folder in which the CSV file is to be created.
- data_export_filename: The output file name. If missing or empty, a sanitized version of the chart name is used.

Note that these properties come from two sources to allow meaningful batch export. export_data and image_export_folder come from the export dialog because they are common to all charts, and image_export_filename comes from the chart properties because it is specific to each chart.

Parameters:

- df: the dataframe to save
- props *(dict)*: the properties

get_data_export_filepath()

```
get_data_export_filepath(props)
```

Returns the file path for the data to export based on the `data_export_format`, `data_export_folder` and `data_export_filename` properties given in `props`. If a relative filename is returned, it is relative to the working directory when the data export takes place.

histogram_bin_edges()

```
histogram_bin_edges(values, bins=None, range=None, weights=None)
```

An improved version of numpy.histogram_bin_edges. This will only return integer edges for input arrays consisting entirely of integers (unless the `bins` are explicitly given otherwise). In addition, the rightmost edge will always be strictly greater than the maximum of `values` (unless explicitly given otherwise in `range`).

confidence_interval()

```
confidence_interval(alpha, data)
```

Returns the half-length of the confidence interval of the mean of `data`, assuming normal distribution, for the given confidence level `alpha`.

Parameters:

- `alpha` *(float)*: Confidence level, must be in the [0..1] range.
- `data` *(array-like)*: An array containing the values.

pivot_for_barchart()

```
pivot_for_barchart(df, groups, series, confidence_level=None)
```

Turns a DataFrame containing scalar results (in the format returned by `results.get_scalars()`) into a 3-tuple of a value, an error, and a metadata DataFrame, which can then be passed to `utils.plot_bars()`. The error dataframe is None if no confidence level is given.

Parameters:

- `df` *(pandas.DataFrame)*: The dataframe to pivot.
- `groups` *(list)*: A list of column names, the values in which will be used as names for the bar groups.
- `series` *(list)*: A list of column names, the values in which will be used as names for the bar series.
- `confidence_level` *(float, optional)*: The confidence level to use when computing the sizes of the error bars.

Returns:

- A triplet of DataFrames containing the pivoted data: (values, errors, metadata)

pivot_for_scatterchart()

```
pivot_for_scatterchart(df, xaxis_itervar, group_by, confidence_level=None)
```

Turns a DataFrame containing scalar results (in the format returned by `results.get_scalars()`) into a DataFrame which can then be passed to `utils.plot_lines()`.

Parameters:

- `df` *(pandas.DataFrame)*: The dataframe to pivot.
- `xaxis_itervar` *(string)*: The name of the iteration variable whose values are to be used as X coordinates.
- `group_by` *(list)*: A list of column names, the values in which will be used to group the scalars into lines.
- `confidence_level` *(float, optional)*: The confidence level to use when computing the sizes of the error bars.

Returns:

- A DataFrame containing the pivoted data, with these columns: `name`, `x`, `y`, and optionally `error` - if `confidence_level` is given.

get_confidence_level()

```
get_confidence_level(props)
```

Returns the confidence level from the `confidence_level` property, converted to a `float`. Also accepts "none" (returns `None` in this case), and percentage values (e.g. "95%").

perform_vector_ops()

```
perform_vector_ops(df, operations:  str)
```

Performs the given vector operations on the dataframe, and returns the resulting dataframe. Vector operations primarily affect the `vectime` and `vecvalue` columns of the dataframe, which are expected to contain `ndarray`'s of matching lengths.

`operations` is a multiline string where each line denotes an operation; they are applied in sequence. The syntax of one operation is:

[(`compute`|`apply`) `:`] *opname* [`(` *arglist* `)`] [`#` *comment*]

Blank lines and lines only containing a comment are also accepted.

opname is the name of the function, optionally qualified with its package name. If the package name is omitted, `omnetpp.scave.vectorops` is assumed.

`compute` and `apply` specify whether the newly computed vectors will replace the input row in the DataFrame (*apply*) or added as extra lines (*compute*). The default is *apply*.

See the contents of the `omnetpp.scave.vectorops` package for more information.

set_plot_title()

```
set_plot_title(title, suggested_chart_name=None)
```

Sets the plot title. It also sets the suggested chart name (the name that the IDE offers when adding a temporary chart to the Analysis file.)

fill_missing_titles()

```
fill_missing_titles(df)
```

Utility function to fill missing values in the `title` and `moduledisplaypath` columns from the `name` and `module` columns. (Note that `title` and `moduledisplaypath` normally come from result attributes of the same name.)

extract_label_columns()

```
extract_label_columns(df, props)
```

Utility function to make a reasonable guess as to which column of the given DataFrame is most suitable to act as a chart title and which ones can be used as legend labels.

Ideally a "title column" should be one in which all lines have the same value, and can be reasonably used as a title. This is often the `title` or `name` column.

Label columns should be a minimal set of columns whose corresponding value tuples uniquely identify every line in the DataFrame. These will primarily be iteration variables and run attributes.

Returns:

A pair of a string and a list; the first value is the name of the "title" column, and the second one is a list of pairs, each containing the index and the name of a "label" column.

Example: `('title', [(8, 'numHosts'), (7, 'iaMean')])`

make_chart_title()

```
make_chart_title(df, title_cols)
```

Produces a reasonably good chart title text from a result DataFrame, given a selected list of "title" columns.

select_best_partitioning_column_pair()

```
select_best_partitioning_column_pair(df, props=None)
```

Choose two columns from the dataframe which best partitions the rows of the dataframe, and returns their names as a pair. Returns (`None`, `None`) if no such pair was found. This method is useful for creating e.g. a bar plot.

select_groups_series()

```
select_groups_series(df, props)
```

Extracts the column names to be used for groups and series from the `df` DataFrame, for pivoting. The columns whose names are to be used as group names are given in the "groups" property in `props`, as a comma-separated list. The names for the series are selected similarly, based on the "series" property. There should be no overlap between these two lists.

If both "groups" and "series" are given (non-empty), they are simply returned as lists after some sanity checks. If both of them are empty, a reasonable guess is made for which columns should be used, and (["module"], ["name"]) is used as a fallback.

The data in `df` should be in the format as returned by `result.get_scalars()`, and the result can be used directly by `utils.pivot_for_barchart()`.

Returns: - (group_names, series_names): A pair of lists of strings containing the selected names for the groups and the series, respectively.

select_xaxis_and_groupby()

```
select_xaxis_and_groupby(df, props)
```

Extracts an iteration variable name and the column names to be used for grouping from the `df` DataFrame, for pivoting. The columns whose names are to be used as group names are given in the "group_by" property in `props`, as a comma-separated list. The name of the iteration variable is selected similarly, from the "xaxis_itervar" property. The "group_by" list should not contain the given "xaxis_itervar" name.

If both "xaxis_itervar" and "group_by" are given (non-empty), they are simply returned after some sanity checks, with "group_by" split into a list. If both of them are empty, a reasonable guess is made for which columns should be used.

The data in `df` should be in the format as returned by `result.get_scalars()`, and the result can be used directly by `utils.pivot_for_scatterchart()`.

Returns: - (xaxis_itervar, group_by): An iteration variable name, and a list of strings containing the selected column names to be used as groups.

assert_columns_exist()

```
assert_columns_exist(df, cols, message="Expected column missing from
DataFrame")
```

Ensures that the dataframe contains the given columns. If any of them are missing, the function raises an error with the given message.

Parameters:

- `cols` *(list of strings)*: Column names to check.

to_numeric()

```
to_numeric(df, columns=None, errors="ignore", downcast=None)
```

Convenience function. Runs `pandas.to_numeric` on the given (or all) columns of `df`. If any of the given columns doesn't exist, throws an error.

Parameters:

- `df` *(DataFrame)*: The DataFrame to operate on
- `columns` *(list of strings)*: The list of column names to convert. If not given, all columns will be converted.
- `errors`, `downcast` *(string)*: Will be passed to `pandas.to_numeric()`

parse_rcparams()

```
parse_rcparams(rc_content)
```

Accepts a multiline string that contains rc file content in Matplotlib's RcParams syntax, and returns its contents as a dictionary. Parse errors and duplicate keys are reported via exceptions.

make_fancy_xticklabels()

```
make_fancy_xticklabels(ax)
```

Only useful for Matplotlib plots. It causes the x tick labels to be rotated by the minimum amount necessary so that they don't overlap. Note that the necessary amount of rotation typically depends on the horizontal zoom level.

split()

```
split(s, sep=",")
```

Split a string with the given separator (by default with comma), trim the surrounding whitespace from the items, and return the result as a list. Returns an empty list for an empty or all-whitespace input string. (Note that in contrast, `s.split(',')` will return an empty array, even for `s=''`.)

L.1.5 Module omnetpp.scave.vectorops

Contains operations that can be applied to vectors.

In the IDE, operations can be applied to vectors on a vector chart by means of the plot's context menu and by editing the *Vector Operations* field in the chart configuration dialog.

Every vector operation is implemented as a function. The notation used in the documentation of the individual functions is:

- *y[k]*: The kth value in the input
- *t[k]*: The kth timestamp in the input

- *yout[k]*: The kth value in the output
- *tout[k]*: The kth timestamp in the output

A vector operation function accepts a DataFrame row as the first positional argument, and optionally additional arguments specific to its operation. When the function is invoked, the row will contain a `vectime` and a `vecvalue` column (both containing NumPy `ndarray`'s) that are the input of the operation. The function should return a similar row, with updated `vectime` and a `vecvalue` columns.

Additionally, the operation may update the `name` and `title` columns (provided they exist) to reflect the processing in the name. For example, an operation that computes *mean* may return `mean(%s)` as name and `Mean of %s` as title (where `%s` indicates the original name/title).

The `aggregate()` and `merge()` functions are special. They receive a DataFrame instead of a row in the first argument, and return new DataFrame with the result.

Vector operations can be applied to a DataFrame using `utils.perform_vector_ops(df,ops)`. `ops` is a multiline string where each line denotes an operation; they are applied in sequence. The syntax of one operation is:

[(`compute`|`apply`) `:`] *opname* [`(` *arglist* `)`] [`#` *comment*]

opname is the name of the function, optionally qualified with its package name. If the package name is omitted, `omnetpp.scave.vectorops` is assumed.

`compute` and `apply` specify whether the newly computed vectors will replace the input row in the DataFrame (*apply*) or added as extra lines (*compute*). The default is *apply*.

To register a new vector operation, define a function that fulfills the above interface (e.g. in the chart script, or an external `.py` file, that the chart script imports), with the `omnetpp.scave.vectorops.vector_operation` decorator on it.

Make sure that the registered function does not modify the data of the NumPy array instances in the rows, because it would have an unwanted effect when used in `compute` (as opposed to `apply`) mode.

Example:

```
from omnetpp.scave import vectorops

@vectorops.vector\_operation("Fooize", "foo(42)")
def foo(r, arg1, arg2=5):
    \# r.vectime = r.vectime * 2    \# <- this is okay
    \# r.vectime *= 2               \# <- this is NOT okay!

    r.vectime = r.vectime * arg1 + arg2
    if "title" in r:
        r.title = r.title + ", but fooized" \# this is also okay
    return r
```

perform_vector_ops()

```
perform_vector_ops(df, operations:  str)
```

See: utils.perform_vector_ops

vector_operation()

```
vector_operation(label:  str = None, example:  str = None)
```

Returns, or acts as, a decorator; to be used on methods you wish to register as vector operations. Parameters:

- `label`: will be shown on the GUI for the user
- `example`: should be string, containing a valid invocation of the function

Alternatively, this can also be used directly as decorator (without calling it first).

lookup_operation()

```
lookup_operation(module, name)
```

Returns a function from the registered vector operations by name, and optionally module. `module` and `name` are both strings. `module` can also be `None`, in which case it is ignored.

aggregate()

```
aggregate(df, function="average")
```

Aggregates several vectors into a single one, aggregating the y values *at the same time coordinate* with the specified function. Possible values: 'sum', 'average', 'count', 'maximum', 'minimum'

merge()

```
merge(df)
```

Merges several series into a single one, maintaining increasing time order in the output.

mean()

```
mean(r)
```

Computes mean on (0,t): yout[k] = sum(y[i], i=0..k) / (k+1).

sum()

```
sum(r)
```

Sums up values: yout[k] = sum(y[i], i=0..k)

add()

```
add(r, c)
```

Adds a constant to all values in the input: yout[k] = y[k] + c

compare()

```
compare(r, threshold, less=None, equal=None, greater=None)
```

Compares value against a threshold, and optionally replaces it with a constant. yout[k] = if y[k] < threshold and less != None then less; else if y[k] == threshold and equal != None then equal; else if y[k] > threshold and greater != None then greater; else y[k] The last three parameters are all independently optional.

crop()

```
crop(r, t1, t2)
```

Discards values outside the [t1, t2] interval. The time values are in seconds.

difference()

```
difference(r)
```

Subtracts the previous value from every value: yout[k] = y[k] - y[k-1]

diffquot()

```
diffquot(r)
```

Calculates the difference quotient of every value and the subsequent one: yout[k] = (y[k+1]-y[k]) / (t[k+1]-t[k])

divide_by()

```
divide_by(r, a)
```

Divides every value in the input by a constant: yout[k] = y[k] / a

divtime()

```
divtime(r)
```

Divides every value in the input by the corresponding time: yout[k] = y[k] / t[k]

expression()

```
expression(r, expression, as_time=False)
```

Replaces the value with the result of evaluating the Python arithmetic expression given as a string: yout[k] = eval(expression). The expression may use the following variables: `t`, `y`, `tprev`, `yprev`, `tnext`, `ynext`, `k`, `n` which stand for t[k], y[k], t[k-1], y[k-1], t[k+1] and y[k+1], k, and the size of vector, respectively.

If `as_time` is `True`, the result will be assigned to the time variable instead of the value variable.

Note that for efficiency, the expression will be evaluated only once, with the variables being `np.ndarray` instances instead of scalar `float` values. Thus, the result is computed using vector operations instead of looping through all vector indices in Python. Expression syntax remains the usual. Most Numpy mathematical functions can be used without module prefix; other Numpy functions can be used by prefixing them with `np.`.

Examples: `2*y+0.5`, `abs(floor(y))`, `(y-yprev)/(t-tprev)`, `fmin(yprev,ynext)`, `cumsum(y)`, `nan_to_num(y)`

integrate()

```
integrate(r, interpolation="sample-hold")
```

Integrates the input as a step function ("sample-hold" or "backward-sample-hold") or with linear ("linear") interpolation.

lineartrend()

```
lineartrend(r, a)
```

Adds a linear component with the given steepness to the input series: yout[k] = y[k] + a * t[k]

modulo()

```
modulo(r, m)
```

Computes floating point reminder (modulo) of the input values with a constant: yout[k] = y[k] % m

movingavg()

```
movingavg(r, alpha)
```

Applies the exponentially weighted moving average filter with the given smoothing coefficient in range (0.0, 1.0]: yout[k] = yout[k-1] + alpha * (y[k]-yout[k-1])

multiply_by()

```
multiply_by(r, a)
```

Multiplies every value in the input by a constant: yout[k] = a * y[k]

removerepeats()

```
removerepeats(r)
```

Removes repeated (consecutive) y values

slidingwinavg()

```
slidingwinavg(r, window_size, min_samples=None)
```

Replaces every value with the mean of values in the window: yout[k] = sum(y[i], i=(k-winsize+1)..k) / winsize If min_samples is also given, allows each window to have only that many valid (not missing [at the ends], and not NaN) samples in each window.

subtractfirstval()

```
subtractfirstval(r)
```

Subtract the first value from every subsequent value: yout[k] = y[k] - y[0]

timeavg()

```
timeavg(r, interpolation)
```

Average over time (integral divided by time), possible parameter values: 'sample-hold', 'backward-sample-hold', 'linear'

timediff()

```
timediff(r)
```

Sets each value to the elapsed time (delta) since the previous value: tout[k] = t[k] - t[k-1]

timeshift()

```
timeshift(r, dt)
```

Shifts the input series in time by a constant (in seconds): tout[k] = t[k] + dt

timedilation()

```
timedilation(r, c)
```

Dilates the input series in time by a constant factor: tout[k] = t[k] * c

timetoserial()

```
timetoserial(r)
```

Replaces time values with their index: tout[k] = k

timewinavg()

```
timewinavg(r, window_size=1)
```

Calculates time average: Replaces the input values with one every 'window_size' interval (in seconds), that is the mean of the original values in that interval. tout[k] = k * winSize, yout[k] = average of y values in the [(k-1) * winSize, k * winSize) interval

timewinthruput()

```
timewinthruput(r, window_size=1)
```

Calculates time windowed throughput: tout[k] = k * winSize, yout[k] = sum of y values in the [(k-1) * winSize, k * winSize) interval divided by window_size

winavg()

```
winavg(r, window_size=10)
```

Calculates batched average: replaces every 'winsize' input values with their mean. Time is the time of the first value in the batch.

L.1.6 Module omnetpp.scave.analysis

This module allows reading, writing, creating and editing OMNeT++ Analysis (.anf) files, querying their contents, and running the charts scripts they contain. The main user of this module is `opp_charttool`.

Class DialogPage

Represents a dialog page in a `Chart`. Dialog pages have an ID, a label (which the IDE displays on the page's tab in the *Chart Properties* dialog), and XSWT content (which describes the UI controls on the page).

```
DialogPage(self, id:  str = None, label:  str = "", content:  str = "")
```

Class Chart

Represents a chart in an `Analysis`. Charts have an ID, a name, a chart script (a Python script that mainly uses Pandas and the `omnetpp.scave.*` modules), dialog pages (which make up the contents of the Chart Properties dialog in the IDE), and properties (which are what the *Chart Properties* dialog in the IDE edits).

```
Chart(self, id:  str = None, name:  str = "", type:  str = "MATPLOTLIB",
template:  str = None, icon:  str = None, script:  str = "",
dialog_pages=[], properties={})
```

Class Folder

Represents a folder in an `Analysis`. Folders may contain charts and further folders.

```
Folder(self, id:  str = None, name:  str = "", items=[])
```

Class Workspace

This is an abstraction of an IDE workspace, and makes it possible to map workspace paths to filesystem paths. This is necessary because the inputs in the `Analysis` are workspace paths.

```
Workspace(self, workspace_dir, project_paths={})
```

Accepts the workspace location, plus a dict that contains the (absolute, or workspace-relative) location of projects by name, for projects that are NOT under the <workspace_dir>/<projectname> location.

```
Workspace.find_workspace(dir=None)
```

Utility function: Find the IDE workspace directory searching from the given directory (or the current dir if not given) upwards. The workspace directory of the Eclipse-based IDE can be recognized by having a `.metadata` subdir.

```
Workspace.get_project_location(self, project_name)
```

Returns the location of the given workspace project in the filesystem path.

```
Workspace.to_filesystem_path(self, wspath)
```

Translate workspace paths to filesystem path.

Class Analysis

Represents an OMNeT++ Analysis, i.e. the contents of an `anf` file. Methods allow reading/writing `anf` files, and running the charts in them for interactive display, image/data export or other side effects.

```
Analysis(self, inputs=[], items=[])
```

```
Analysis.collect_charts(self, folder=None)
```

Collects and returns a list of all charts in the specified folder, or in this Analysis if no folder is given.

```
Analysis.export_data(self, chart, wd, workspace, format="csv", target_folder=None,
filename=None, enforce=True, extra_props={})
```

Runs a chart script for data export. This method just calls `run_chart()` with extra properties that instruct the chart script to perform data export. (It is assumed that the chart script invokes `utils.export_data_if_needed()` or implements equivalent functionality).

```
Analysis.export_image(self, chart, wd, workspace, format="svg", target_folder=None,
filename=None, width=None, height=None, dpi=96, enforce=True, extra_props={})
```

Runs a chart script for image export. This method just calls `run_chart()` with extra properties that instruct the chart script to perform image export. (It is assumed that the chart script invokes `utils.export_image_if_needed()` or implements equivalent functionality).

```
Analysis.from_anf_file(anf_file_name)
```

Reads the given anf file and returns its content as an `Analysis` object.

```
Analysis.get_item_path(self, item)
```

Returns the path of the item (Chart or Folder) within the Analysis as list of path segments (Folder items). The returned list includes both the root folder of the Analysis and the item itself. If the item is not part of the Analysis, None is returned.

```
Analysis.get_item_path_as_string(self, item, separator=" / ")
```

Returns the path of the item (Chart or Folder) within the Analysis as a string. Segments are joined with the given separator. The returned string includes the item name itself, but not the root folder (i.e. for items in the root folder, the path string equals to the item name). If the item is not part of the Analysis, None is returned.

```
Analysis.run_chart(self, chart, wd, workspace, extra_props={}, show=False)
```

Runs a chart script with the given working directory, workspace, and extra properties in addition to the chart's properties. If `show=True`, it calls `plt.show()` if it was not already called by the script.

```
Analysis.to_anf_file(self, filename)
```

Saves the analysis to the given .anf file.

load_anf_file()

```
load_anf_file(anf_file_name)
```

L.1.7 Module omnetpp.scave.charttemplate

Loading and instantiating chart templates.

Class ChartTemplate

Represents a chart template.

```
ChartTemplate(self, id:  str, name:  str, type:  str, icon:  str, script:
str, dialog_pages, properties)
```

Creates a chart template from the given data elements.

Parameters:

- `id` *(string)*: A short string that uniquely identifies the chart template.
- `name` *(string)*: Name of the chart template.
- `type` *(string)*: Chart type, one of: "bar"/"histogram"/"line"/"matplotlib".
- `icon` *(string)*: Name of the icon to be used for charts of this type.
- `script` *(string)*: The Python chart script.
- `dialog_pages` *(list of `DialogPage`)*: Pages of the "Configure Chart" dialog.
- `properties` *(string->string dictionary)*: Initial values for chart properties.

```
ChartTemplate.create_chart(self, id:  int = None, name:  str = None, props=None)
```

Creates and returns a chart object (`org.omnetpp.scave.Chart`) from this chart template. Chart properties will be set to the default values defined by the chart template. If a `props` argument is present, property values in it will overwrite the defaults.

Parameters:

- `id` *(string)*: A numeric string that identifies the chart within the Analysis. Auto-assigned if missing.
- `name` *(string)*: Name for the chart. If missing, the chart template name will be used.
- `props` *(string->string dictionary)*: Chart properties to set. It may not introduce new properties, i.e. the keys must be subset of the property keys defined in the chart template.

get_chart_template_locations()

```
get_chart_template_locations()
```

Returns a list of locations (directories) where the chart templates that come with the IDE can be found.

load_chart_templates()

```
load_chart_templates(dirs=[], add_default_locations=True)
```

Loads chart templates from the given list of directories, and returns them in a dictionary. Chart templates are loaded from files with the `.properties` extension.

Parameters:

- `dirs` *(string list)*: A short string that uniquely identifies the chart template.
- `add_default_locations` *(bool)*: Whether the directories returned by `get_chart_template_locati` should also be searched in addition to the specified directory list.

Returns:

- A string->ChartTemplate dictionary, with chart template IDs used as keys.

load_chart_template()

```
load_chart_template(properties_file)
```

Loads the chart template from the specified `.properties` file, and returns it as a `ChartTemplate` object.

References

[BCH+96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 69–82, New York, NY, USA, 1996. ACM.

[BT00] R. L. Bagrodia and M. Takai. Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages. 11(4):395–414, 2000.

[CM79] M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, (5):440–452, 1979.

[EHW02] K. Entacher, B. Hechenleitner, and S. Wegenkittl. A Simple OMNeT++ Queuing Experiment Using Parallel Streams. *PARALLEL NUMERICS'02 - Theory and Applications*, pages 89–105, 2002. Editors: R. Trobec, P. Zinterhof, M. Vajtersic and A. Uhl.

[EPM99] G. Ewing, K. Pawlikowski, and D. McNickle. Akaroa2: Exploiting Network Computing by Distributing Stochastic Simulation. In *Proceedings of the European Simulation Multiconference ESM'99, Warsaw, June 1999*, pages 175–181. International Society for Computer Simulation, 1999.

[For94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. 8(3/4):165–414, 1994.

[Gol91] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[Hel98] P. Hellekalek. Don't Trust Parallel Monte Carlo. *ACM SIGSIM Simulation Digest*, 28(1):82–89, jul 1998. Author's page is a great source of information, see `http://random.mat.sbg.ac.at/`.

[HPvdL95] Jan Heijmans, Alex Paalvast, and Robert van der Leij. Network Simulation Using the JAR Compiler for the OMNeT++ Simulation System. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, 1991.

[JC85] Raj Jain and Imrich Chlamtac. The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms without Storing Observations. *Communications of the ACM*, 28(10):1076–1085, 1985.

[Kof95] Stig Kofoed. Portable Multitasking in C++. *Dr. Dobb's Journal*, November 1995. Download source from `http://www.ddj.com/ftp/1995/1995.11/mtask.zip/`.

[LAM] LAM-MPI home page. `http://www.lam-mpi.org/`.

[Len94] Gábor Lencse. Graphical Network Editor for OMNeT++. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[LSCK02] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An Objected-Oriented Random-Number Package with Many Long Streams and Substreams. *Operations Research*, 50(6):1073–1075, 2002. Source code can be downloaded from `http://www.iro.umontreal.ca/~lecuyer/papers.html`.

[MN98] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudorandom Number Generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998. Source code can be downloaded from `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[MvMvdW95] André Maurits, George van Montfort, and Gerard van de Weerd. OMNeT++ Extensions and Examples. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[OF00] Hong Ong and Paul A. Farrell. Performance Comparison of LAM/MPI, MPICH and MVICH on a Linux Cluster Connected by a Gigabit Ethernet Network. In *Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, October 10-14, 2000*. The USENIX Association, 2000.

[PFS86] Bratley P., B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, 1986.

[PJL02] K. Pawlikowski, H. Jeong, and J. Lee. On Credibility of Simulation Studies of Telecommunication Networks. *IEEE Communications Magazine*, pages 132–139, jan 2002.

[Pon91] György Pongor. OMNET: An Object-Oriented Network Simulator. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1991.

[Pon92] György Pongor. Statistical Synchronization: A Different Approach of Parallel Discrete Event Simulation. Technical report, University of Technology, Data Communications Laboratory, Lappeenranta, Finland, 1992.

[Pon93] György Pongor. On the Efficiency of the Statistical Synchronization Method. In *Proceedings of the European Simulation Symposium (ESS'93), Delft, The Netherlands, Oct. 25-28, 1993*. International Society for Computer Simulation, 1993.

[Qua] Quadrics home page. `http://www.quadrics.com/`.

[ŞVE03] Y. Ahmet Şekercioğlu, András Varga, and Gregory K. Egan. Parallel Simulation Made Easy with OMNeT++. In *Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands*. International Society for Computer Simulation, 2003.

[Var92] András Varga. OMNeT++ - Portable Simulation Environment in C++. In *Proceedings of the Annual Students' Scientific Conference (TDK), 1992*. Technical University of Budapest, 1992. In Hungarian.

[Var94] András Varga. Portable User Interface for the OMNeT++ Simulation System. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[Var98a] András Varga. K-split – On-Line Density Estimation for Simulation Result Collection. In *Proceedings of the European Simulation Symposium (ESS'98), Nottingham, UK, October 26-28*. International Society for Computer Simulation, 1998.

[Var98b] András Varga. Parameterized Topologies for Simulation Programs. In *Proceedings of the Western Multiconference on Simulation (WMC'98) Communication Networks and Distributed Systems (CNDS'98), San Diego, CA, January 11-14*. International Society for Computer Simulation, 1998.

[Var99] András Varga. Using the OMNeT++ Discrete Event Simulation System in Education. *IEEE Transactions on Education*, 42(4):372, November 1999. (on CD-ROM issue; journal contains abstract).

[Vas96] Zoltán Vass. PVM Extension of OMNeT++ to Support Statistical Synchronization. Master's thesis, Technical University of Budapest, 1996. In Hungarian.

[VF97] András Varga and Babak Fakhamzadeh. The K-Split Algorithm for the PDF Approximation of Multi-Dimensional Empirical Distributions without Storing Observations. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997*, pages 94–98. International Society for Computer Simulation, 1997.

[VP97] András Varga and György Pongor. Flexible Topology Description Language for Simulation Programs. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997*, pages 225–229, 1997.

[VŞE03] András Varga, Y. Ahmet Şekercioğlu, and Gregory K. Egan. A practical efficiency criterion for the null message algorithm. In *Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands*. International Society for Computer Simulation, 2003.

[Wel95] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, 1995.

Index