OMNeT++

User Manual

OMNeT++ version 4.0

Chapters

Table of Contents

---

1 Introduction

1.1 What is OMNeT++?

OMNeT++ 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
- and in general, it can be used for the modeling and simulation of any system where the discrete event approach is suitable, and which can be conveniently mapped into entities communicating by exchanging messages.

OMNeT++ itself is not a simulator of anything concrete, but it 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 may 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, which can be used to customize module behaviour, and/or to parameterize the model's topology. Modules at the lowest level of the module hierarchy are called simple modules, and they encapsulate behaviour. Simple modules are programmed in C++, and make use of the simulation library.

OMNeT++ 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.

OMNeT++ also supports parallel distributed simulation. OMNeT++ 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 plugged in. Models do not need any special instrumentation to be run in parallel -- it is just a matter of configuration. OMNeT++ can even be used for classroom presentation of parallel simulation algorithms, because simulations can be run in parallel even under the GUI which provides detailed feedback on what is going on.

OMNEST is the commercially supported version of OMNeT++. OMNeT++ is only free 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 the following way:

- The chapters [1] and [2] contain introductory material
- The second group of chapters, [3], [4] and [6] are the programming guide. They present the NED language, the simulation concepts and their implementation in OMNeT++, explain how to write simple modules and describe the class library.
- The chapters [10] and [13] elaborate the topic further, by explaining how one can customize the network graphics and how to write NED source code comments from which documentation can be generated.
- The following chapters, [7], [8], [9] and [11] deal with practical issues like building and running simulations and analyzing results, and present the tools OMNeT++ provides to support these tasks.
- Chapter [14] is devoted to the support of distributed execution.
- The chapters [15] and [16] explain the architecture and internals of OMNeT++, as well as ways to extend it and embed it into larger applications.
- The appendices provide a reference of the NED language, configuration options, file formats and other details.

1.3 Credits

OMNeT++ has been developed by András Varga (andras@omnetpp.org, andras.varga@omnest.com).

In the early stage of the project, several people have contributed to OMNeT++. Although most contributed code is no longer part of the OMNeT++, nevertheless I'd like to acknowledge the work of the following people. First of all, I'd like thank Dr Gy?rgy Pongor (pongor@hit.bme.hu), my advisor at the Technical University of Budapest who initiated the OMNeT++ as a student project.

My fellow student ákos Kun started to program the first NED parser in 1992-93, but it was abandoned after a few months. The first version of nedc was finally developed in summer 1995, by three exchange students from TU Delft: Jan Heijmans, Alex Paalvast and Robert van der Leij. nedc was first called JAR after their initials until it got renamed to nedc. nedc was further developed and refactored several times until it finally retired and got replaced by nedtool in OMNeT++ 3.0. The second group of Delft exchange students (Maurits André, George van Montfort, Gerard van de Weerd) arrived in fall 1995. They performed some testing of the simulation library, and wrote some example simulations, for example the original version of Token Ring, and simulation of the NIM game which survived until OMNeT++ 3.0. These student exchanges were organized by Dr. Leon Rothkranz at TU Delft, and Gy?rgy Pongor at TU Budapest.

The diploma thesis of Zoltán Vass (spring 1996) was to prepare OMNeT++ for parallel execution over PVM to OMNeT++. This code has been replaced with the new Parallel Simulation Architecture in OMNeT++ 3.0. Gábor Lencse (lencse@hit.bme.hu) was also interested in parallel simulation, namely a method called Statistical Synchronization (SSM). He implemented the FDDI model (practically unchanged until now), and added some extensions into NED for SSM. These extensions have been removed since then (OMNeT++ 3.0 does parallel execution on different principles).

The P^2 algorithm and the original implementation of the k-split algorithm was programmed in fall 1996 by Babak Fakhamzadeh from TU Delft. k-split was later reimplemented by András.

Several bugfixes and valuable suggestions for improvements came from the user community of OMNeT++. It would be impossible to mention everyone here, and the list is constantly growing -- instead, the README and ChangeLog files contain acknowledgements.

Between summer 2001 and fall 2004, the OMNeT++ CVS was hosted at the University of Karlsruhe. Credit for setting up and maintaining the CVS server goes to Ulrich Kaage. Ulrich can also be credited with converting the User Manual from Microsoft Word format to LaTeX, which was a huge undertaking and great help.

---

2 Overview

2.1 Modeling concepts

An OMNeT++ 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 not limited. The whole model, called network in OMNeT++, is itself a compound module. Messages can be sent either via connections that span between modules or directly to other modules. The concept of simple and compound modules is similar to DEVS atomic and coupled models. In Fig. below, boxes represent simple modules (gray background) and compound modules. Arrows connecting small boxes represent connections and gates.

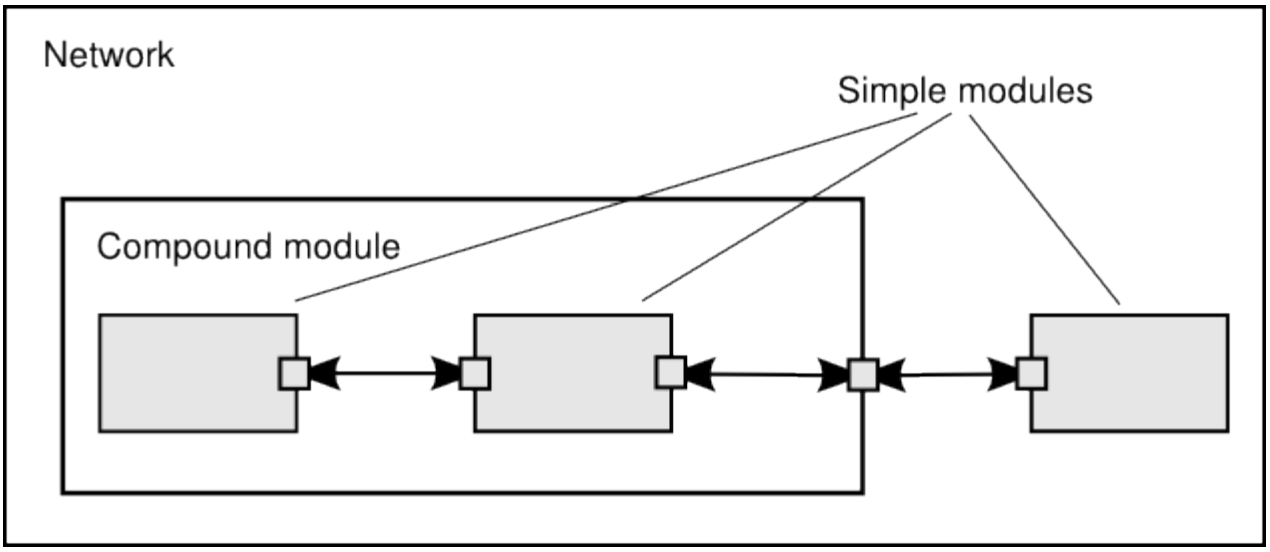


Figure: Simple and compound modules

Modules communicate with messages which -- in addition to usual attributes such as timestamp -- may contain arbitrary data. 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 out through output gates and arrive through input gates. An input and an output gate can be linked with 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 across hierarchy levels are not permitted, as it would hinder model reuse. Due to the hierarchical structure of the model, messages typically travel through a chain of connections, to start and arrive in simple modules. Compound modules act as 'cardboard boxes' in the model, transparently relaying messages between their inside 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 mainly used to pass configuration data to simple modules, and to help define model topology. Parameters may 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.

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

- 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 OMNeT++ model consists of hierarchically nested modules, which communicate by passing messages to each another. OMNeT++ models are often referred to as *networks*. The top level module is the *system module*. The system module contains *submodules*, which can also contain submodules themselves (Fig. below). The depth of module nesting is not limited; this allows the user to reflect the logical structure of the actual system in the model structure.

Model structure is described in OMNeT++'s NED language.

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

2.1.2 Module types

Both simple and compound modules are instances of *module types*. While 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, there is no distinction whether it is a simple or a compound module. This allows the user to split a simple module into several simple modules embedded into a compound module, or vica versa, 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 [9].

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. below).

Due to the hierarchical structure of the model, messages typically travel through a series of connections, to start and arrive in simple modules. Such series of connections that go from simple module to simple module are called *routes*. Compound modules act as `cardboard boxes' in the model, transparently relaying messages between their inside and the outside world.

2.1.4 Modeling of packet transmissions

Connections can be assigned three parameters, which facilitate the modeling of communication networks, but can be useful in other models too: *propagation delay*, *bit error rate* and *data rate*, all three being optional. One can specify link parameters individually for each connection, or define link types and use them throughout the whole model.

Propagation delay is the amount of time the arrival of the message is delayed by when it travels through the channel.

Bit error rate specifies the probability that a bit is incorrectly transmitted, and allows for simple noisy channel modelling.

Data rate is specified in bits/second, and it is used for calculating transmission time of a packet.

When data rates are in use, the sending of the message in the model corresponds to the transmission of the first bit, and the arrival of the message corresponds to the reception of the last bit. This model is not always applicable, for example protocols like Token Ring and FDDI do not wait for the frame to arrive in its entirety, but rather start repeating its first bits soon after they arrive -- in other words, frames ``flow through'' the stations, being delayed only a few bits. If you want to model such networks, it is possible to change this default behaviour and deliver the message to the module when the first bit is received.

2.1.5 Parameters

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

Parameters may be used to customize simple module behaviour, and for parameterizing 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 OMNeT++ simulation class library. The simulation programmer can choose between event-driven and process-style description, and can 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:

- modules, gates, connections etc.
- parameters
- messages
- container classes (e.g. queue, array)
- data collection classes
- statistic and distribution estimation classes (histograms, P^2 algorithm for calculating quantiles etc.)
- transient detection and result accuracy detection classes

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 has made it possible to create a simulation GUI where all internals of the simulation are visible.

2.3 Using OMNeT++

2.1.1 Building and running simulations

This section provides insight into working with OMNeT++ in practice: Issues such as model files, compiling and running simulations are discussed.

An OMNeT++ model consists of the following parts:

- NED language topology description(s) (`.ned` files) which describe the module structure with parameters, gates etc. NED files can be written using any text editor, but the OMNeT++ IDE provides excellent support for two-way graphical and text editing.
- Message definitions (`.msg` files). You can define various message types and add data fields to them. OMNeT++ will translate message definitions into full-fledged C++ classes.
- Simple modules 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. OMNeT++ 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, thus it can be run on other machines without OMNeT++ being present or it can be created as a shared library. In this case the OMNeT++ shared libraries must be present on that system. When the program is started, first it reads all NED files containing your model topology, then it reads a configuration file (usually called `omnetpp.ini`). This 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 data files: output vector files, output scalar files , and possibly the user's own output files. OMNeT++ contains an Integrated Development Environment that provides rich environment for analyzing these files. It is not expected that someone will process the result files using OMNeT++ alone: output files are text files in a format which can be read into math packages like Matlab or Octave, or imported into spreadsheets like OpenOffice Calc, Gnumeric or MS Excel (some preprocessing using `sed`, `awk` or `perl` might be required, this will be discussed later). All these external programs provide rich functionality for statistical analysis and visualization, and it is outside the scope of OMNeT++ to duplicate their efforts. This manual briefly describes some data plotting programs and how to use them with OMNeT++.

Output scalar files can be visualized with the OMNeT++ IDE too. It can draw bar charts, x-y plots (e.g. throughput vs offered load), or export data via the clipboard for more detailed analysis into spreadsheets and other 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. Just as important, a hands-on experience allows the user to get a `feel' of the model's behaviour. The graphical user interface can also be used to demonstrate a model's operation.

The same simulation model can be executed with different user interfaces, without any change in the model files themselves. The user would test and debug the simulation with a powerful graphical user interface, and finally run it with a simple and fast user interface that supports batch execution.

Component libraries

Module types can be stored in files separate from the place of their actual use. This enables 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

If you installed the source distribution, the omnetpp directory on your system should contain the following subdirectories. (If you installed a precompiled distribution, some of the directories may be missing, or there might be additional directories, e.g. containing software bundled with OMNeT++.)

The simulation system itself:

```
omnetpp/          OMNeT++ root directory
  bin/            OMNeT++ executables
  include/        header files for simulation models
  lib/            library files
  images/         icons and backgrounds for network graphics
  doc/            manuals, readme files, license, APIs, etc.
    manual/       manual in HTML
    migration/    how to migrate your models from 3.x to 4.0 version
    ned2/         DTD definition of the XML syntax for NED files
    tictoc-tutorial/  introduction into using OMNeT++
    api/          API reference in HTML
    nedxml-api/   API reference for the NEDXML library
    parsim-api/   API reference for the parallel simulation library
  migrate/        tools to help model migration from 3.x to 4.0 version
  src/            OMNeT++ 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
    tkenv/        Tcl/Tk-based user interface
    nedxml/       NEDXML library, nedtool, opp_msgc
    scave/        result analysis library
    eventlog/     eventlog processing library
    layout/       graph layouter for network graphics
    common/       common library
    utils/        opp_makemake, opp_test, etc.
  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 OMNeT++ 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 OMNeT++.

```
    mingw/        MinGW gcc port
    msys/         MSYS plus libraries
```

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 OMNeT++ community.

```
    contrib/      directory for contributed material
      octave/     Octave scripts for result processing
      emacs/      NED syntax highlight for Emacs
      ...
```

---

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*, 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 via introducing hierarchies. In OMNeT++, 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`) was also 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 system like SQL databases, and so on.

> **NOTE**
> This chapter is going to explain the NED language gradually, via examples. If you are looking for a more formal and concise treatment, see Appendix [18].

3.2 Warmup

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. One each node there's 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 below.

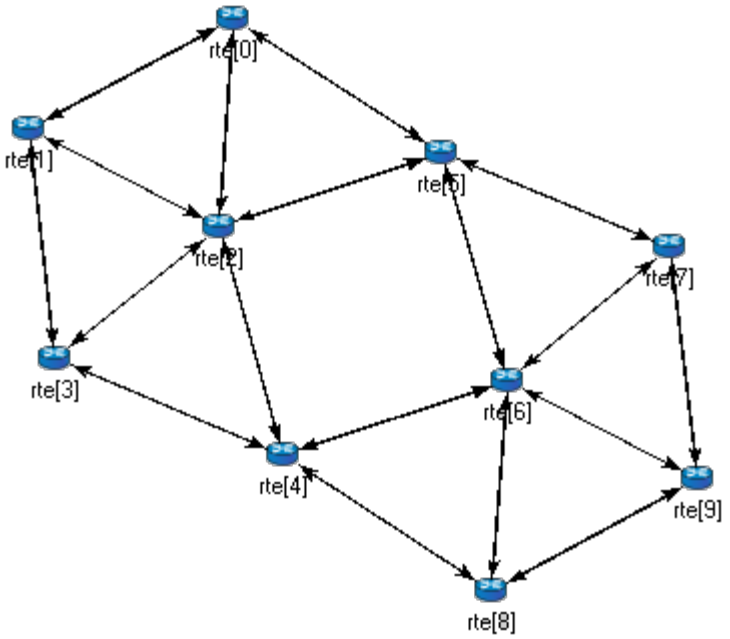


Figure: The network

The corresponding NED description would look like this:

```
//
// A network
//
network 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 OMNeT++ IDE they also get 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 [13].

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 OMNeT++ 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 he wants 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 details.

3.2.3 The App, Routing and Queue simple modules

Simple modules are the basic building blocks for other (compound) modules. 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 see 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.

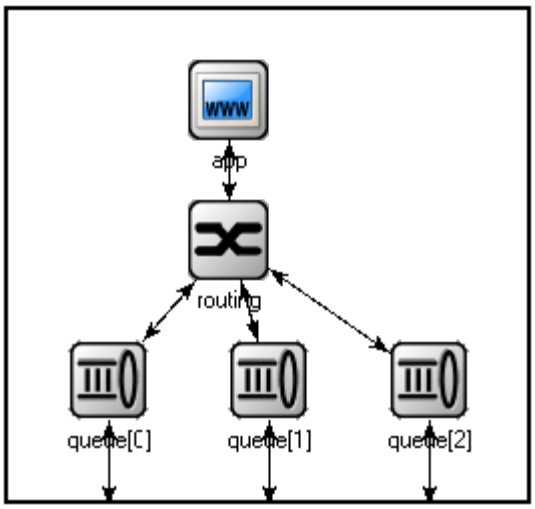


Figure: The Node compound module

```
module Node
{
    parameters:
        @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 seen all NED definitions, but how does it get used by OMNeT++? 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 if the executable or gets loaded from 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 gets 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 vector sizes get assigned and they get fully connected before proceeding to go into the submodules to build their internals.

* * *

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

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's 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, OMNeT++ 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 you have several modules that are all in a common namespace, then 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 {
   ...
}
```

As you've seen, `@namespace` can be specified on file level. Moreover, when placed in a file called `package.ned`, the namespace will apply to all files in the same directory and all directories 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 (no C++ code) is associated with it.

> **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.

Module and channel types only used locally can be defined in the `types` section as inner types, so that they don't 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" submodules or connections, or to modify inherited ones.

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

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

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

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

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

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 observed), 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  // needs a CustomChannel C++ class
{
}
```

The practical difference to modules is that you rarely need to write you own channel C++ class, because there are predefined channel types that you can subclass from, inheriting their C++ code. The predefined types are: `ned.IdealChannel`, `ned.DelayChannel` and `ned.DatarateChannel`. (``ned'' is the package name; you can get rid of it if you import the types with the `import ned.*` or similar 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 bandwidth of the channel, and it is used for calculating the transmission duration of packets. Values need to be specified with 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.
- `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 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 would decide 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 C 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 not explicitly specified.

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

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

Parameters are primarily useful as input to the underlying C++ class, but even if you reuse the underlying C++ class of built-in channel types, they may be read and used by other parts of the model. For example, adding a `cost` parameter (or `@cost` property) may be observed by the routing algorithm and used for routing decisions. The following example shows a `cost` parameter, and annotation using a property (`@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`** and **`xml`**; they can also be declared **`volatile`**. For the numeric types, a unit of measurement can also be specified (`@unit` property), to increase type safety.

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

Let us see an example before we go into details:

```
simple App
{
    parameters:
        int address;  // local node address
        string destAddresses;  // destination addresses
        volatile double sendIaTime @unit(s) = default(exponential(1s));
                                // time between generating packets
        volatile int packetLength @unit(byte);  // length of one packet
    ...
}
```

Values

Parameters may get their values from several places: from NED code, from the configuration (`omnetpp.ini`), or even, interactively from the user.

The following example shows how parameters of an `App` module (from the previous example) may be assigned when `App` gets used as a submodule:

```
module Node
{
    submodules:
        app : App {
            sendIaTime = 3s;
            packetLength = 1024B; // B=byte
        }
        ...
}
```

After the above definition, the `app` submodule's parameters cannot be changed from `omnetpp.ini` any more.

> **IMPORTANT**
> A value assigned in NED cannot be overwritten from ini files; they count as "hardcoded" as far as ini files are concerned.

Provided that the value isn't set in the NED file, a parameter can be assigned in the configuration in the following way:

```
**.sendIaTime = 100ms
```

The above line applies to all parameters called `sendIaTime`, whichever module they belong to; it is possible to write more selective assignments by replacing `**` with more specific patterns. Parameter assignments in the configuration are described in section [8.4].

One can also write expressions, including stochastic expressions, in the ini file:

```
**.sendIaTime = 2s + exponential(100ms)
```

If there is no assignment in the ini file, the default value (given with `=default(...)` in NED) is 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):

```
**.sendIaTime = 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):

```
**.sendIaTime = ask
```

> **NOTE**
> How do you decide whether to assign a parameter from NED or from an ini file? The point of ini files is 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.

Expressions

Parameter values may be given with expressions. NED language expressions have a C-like syntax, with 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. Expressions can use various numeric, string, stochastic and other functions (`fabs()`, `toUpper()`, `uniform()`, `erlang_k()`, etc.).

NOTE
The list of NED functions can be read in Appendix [20]. The user can also extend NED with new functions; this feature is described in XXX.

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

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

volatile

The **`volatile`** modifier causes the parameter's value expression to be evaluated every time the parameter is read. This has significance if the expression is not constant, for example it involves numbers drawn from a random number generator. In contrast, non-volatile parameters are evaluated only once. (This practically means that they are evaluated and replaced with the resulting constant at the start of the simulation.)

To better understand **`volatile`**, let's suppose we have an `ActiveQueue` simple module that has a `volatile double` parameter named `serviceTime`.

The queue module's C++ implementation would re-read the `serviceTime` parameter at runtime for every job serviced; so if `serviceTime` is assigned an expression like `uniform(0.5s, 1.5s)`, every job would have a different, random service time.

In practice, a volatile parameter usually means that the underlying C++ code will re-read the parameter every time a value is needed at runtime, so the parameter can be used a source of random numbers.

NOTE
This does not mean that a non-volatile parameter cannot be assigned a value like `uniform(0.5s, 1.5s)`. It can, but that has a totally different effect. Had we omitted the `volatile` keyword from the `serviceTime` parameter, for example, the effect would be that every job had the *same* constant service time, say `1.2975367s`, chosen randomly at the beginning of the simulation.

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 sendIaTime @unit(s) = default(exponential(350ms));
        volatile int packetLength @unit(byte) = default(4KB);
    ...
}
```

The `@unit(s)` and `@unit(byte)` bits declare 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 OMNeT++ is listed in the Appendix, see [17.5.7]. Unknown units (`bogomips`, etc.) can also be used, but there are no conversions for them, i.e. decimal prefixes will not be recognized.

The OMNeT++ 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.

XML parameters

Sometimes modules need more complex input than simple module parameters can describe. Then you'd put these parameters into an external config file, and let the modules read and process the file. You'd pass the file name to the modules in a string parameter.

These days, XML is increasingly becoming a standard format for configuration files as well, so you might as well describe your configuration in XML. From the 3.0 version, OMNeT++ contains built-in support for XML config files.

OMNeT++ wraps the XML parser (LibXML, Expat, etc.), reads and DTD-validates the file (if the XML document contains a DOCTYPE), caches the file (so that if you refer to it from several modules, it'll still be loaded only once), lets you pick parts of the document via an XPath-subset notation, and presents the contents to you in a DOM-like object tree.

This machinery can be accessed via the NED parameter type `xml`, and the `xmldoc()` operator. You can point `xml`-type module parameters to a specific XML file (or to an element inside an XML file) via the `xmldoc()` operator. You can assign `xml` parameters both from NED and from `omnetpp.ini`.

The following example declares an `xml` parameter, and assigns an XML file to it:

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

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

It is also possible to assign an XML element within a file to the parameter:

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

```
<?xml>
XXX example
```

3.7 Gates

Gates are the connection points of modules. OMNeT++ 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, cannot be connected to two or more other gates. (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.

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 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, you can annotate selected gates with the `@loose` property, which turns off connectivity check for that gate. Also, input gates that solely exist so that the module can receive messages via `sendDirect()` (see [4.6.6]) should be annotated with `@directIn`. It is also possible to turn off 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 vector, so that it can be connected to several modules:

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

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 may be given explicitly, but, as we'll see later, it is also possible to specify the type with a string expression (see section [3.11].)

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

The basic syntax of submodules is shown below:

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

A submodule vector may also be used to implement a conditional submodule, like in the example below:

```
module Host
{
    parameters:
        bool withTCP = default(true);
    submodules:
        tcp[withTCP ? 1 : 0]: TCP;  // conditional submodule
        ...
    connections:
        tcp[0].ipOut --> ip.tcpIn if withTCP;
        tcp[0].ipIn <-- ip.tcpOut if withTCP;
        ...
}
```

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. This is described in chapter [10].

```
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;  // different "id" parameter for each element
        }
    connections:
        ...
}
```

An empty body may be omitted, that is,

```
      queue: Queue;
```

is the same as

```
      queue: Queue {
      }
```

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

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). 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 Warmup 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 on 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 size for the two directions.

Channel specification

A channel specification (`--> channelspec -->` inside a connection) are similar to submodules in many respect.

Let's see some examples:

```
<--> {delay=10ms;} <-->
<--> {delay=10ms; datarate=1e-8;} <-->
<--> C <-->
<--> BBone {cost=100; length=52km; datarate=1e-8;} <-->
<--> {@display("XXX");} <-->
<--> BBone {@display("XXX");} <-->
```

When a channel type is missing, one of the built-in channel types will be used, based on the parameters assigned in the connection. 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 an connection without an explicit channel type, it will be an error (with ``*ned.DelayChannel has no such parameter*'' or similar message).

3.10 Multiple connections

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

This will be shown in the following 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 a gate which is not leading anywhere.

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, 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 given by the network setup code for unconnected gates.

3.10.1 Connection patterns

Several approaches can be used when you want to create complex topologies which 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, 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 vica versa.

Though this pattern is very general, its usage can be prohibitive if the *N* number of nodes is high and the graph is sparse (it has much fewer connections that N^2). 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, 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[...];
}
```

The 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, you can resort to listing all connections, like you would do it in most existing simulators.

3.11 Submodule type as parameter

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="SensorNode"`, then the module vector will consist of sensor nodes (provided such module type exists and it qualifies -- the latter will be explained right now).

The missing piece is the `like INode` bit. `INode` must be an existing *module interface*, which the `SensorNode` module type must implement (more about this later).

The corresponding NED declarations:

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

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

3.12 Properties (metadata annotations)

Properties allow adding metadata annotations to modules, parameters, gates, connections, NED files, packages, and virtually anything in NED. `@display`, `@class`, `@namespace`, `@unit`, `@prompt`, `@loose`, `@directIn` are all properties that have been mentioned in previous sections, but those examples only scratch the surface of what can be done with properties.

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 you cannot have properties per-instance different properties. 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:

```
@prop;  // file property

module Example
{
    parameters:
       @prop;   // module property
       int a @prop = default(1); // parameter property
    gates:
       output out @prop;
```

```
    submodules:
       src: Source {
           parameters:
              @prop;  // submodule property
              count @prop;  // adding a property to a parameter
           gates:
              out[] @prop;  // adding a property to a gate
       }
       ...
    connections:
       src.out++ --> { @prop; } --> sink1.in;
       src.out++ --> Channel { @prop; } --> sink2.in;
}
```

Data model

Properties may contain data, given in parentheses; the data model is quite flexible. Properties may contain lists:

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

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

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

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

```
@nodeinfo(id=742;labels=swregion,routers,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. `@prop` is the same as `@prop()`.

The syntax for value items is a bit restrictive: they may contain words, numbers, string constants and some more, but not arbitrary strings. Whenever the syntax does not permit some value, it should be enclosed in quotes. This quoting does not make any difference in the value, because the parser automatically drops one layer of quotes; thus, `@class(TCP)` and `@class("TCP")` are exactly the same.

There are also some conventions. One can use properties to tag some NED element with a label; for example, a `@host` property could be used to mark all module types that represent various hosts. This property could be used e.g. by editing tools, by topology discovery code inside the simulation model, etc.

The convention for such a "label" property is that any extra data in it (i.e. within parens) is ignored, except a single word `false`, which is reserved to "remove" the property. Thus, simulation model or tool source code 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.

Modifying properties

When you subclass a NED type, use a module type as submodule or use a channel type for a connection, you may add new properties to the module or channel, or to its parameters and gates, and you can also modify existing properties.

When modifying a property, the new property gets merged with the old one, with a few simple rules. 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 [10]).

Indices

Properties are identified by names, so if the same *@name* occurs in the same context, then it names the exact same property object. If you want to have multiple properties with the same name, then you need to distinguish them with an index. An index is a name or number, written in square brackets after the property name. The index may be chosen to carry a meaning, or it may be a dummy whose only purpose is to tell multiple properties with the same name apart. (The code that interprets properties may be written to observe or to ignore indices, as needed).

The following example, the simple module declares in properties the statistics it collects. These declarations might be used by model editing tools, by simulation code, or by analysis tools.

[Note that this is a completely hypothetical example -- OMNeT++ does not presently support declaring statistics using properties.]

The statistic names are used as indices:

```
simple App {
    // declare two statistics collected by the C++ code
    @statistic[packetsReceived](type=integer;label="Number of packets received");
    @statistic[responseTime](type=double;unit=s;label="Application response
time");
    ...
}

simple HttpApp extends App {
    // tailor the label of the "packetsReceived" statistic:
    @statistic[packetsReceived](label="Number of HTTP requests received");
}
```

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 only extend a simple module, compound module may only extend a compound module, and so on. 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 your 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]).

3.14 Packages

Small simulation projects are fine to have all NED files in a single directory. When a project grows, however, it sooner or later becomes inevitable 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 clashes, 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.

Overview

When a simulation is run, you must tell the simulation kernel the directory which is the root of your 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. You can have several NED directory trees, and their roots (the NED source folders) should be given to the simulation kernel in the NEDPATH variable. NEDPATH 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. NEDPATH is described in details in chapter [9].

Directories in a NED source tree correspond to packages. If you have NED files in a `<root>/a/b/c` directory (where `<root>` gets listed in NEDPATH), 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. That is, if you have a `package.ned` file in your `<root>` directory whose package declaration says `org.example.myproject`, then the `<root>/a/b/c` directory will be package `org.example.myproject.a.b.c` -- and NED files in them must contain that as package declaration. Only the root `package.ned` has this property, other `package.ned`'s cannot change the package.

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;
```

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 you can refer to an existing

type:

- By fully qualified name. This is often cumbersome though, as names tend to be too long;
- Import the type, then the simple name will be enough;
- 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's 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.

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] 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's none or there's 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`; and suppose that there's also 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's an error because there're 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"`.

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*.

---

4 Simple Modules

Simple modules are the active components in the model. Simple modules are programmed in C++, using the OMNeT++ class library. The following sections contain a short introduction to discrete event simulation in general, explain how its concepts are implemented in OMNeT++, 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 OMNeT++ 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 (in contrast to *continuous* systems where state changes are continuous). Those systems that can be viewed as Discrete Event Systems can be modeled using Discrete Event Simulation. (Other systems can be modelled e.g. with continuous simulation models.)

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 person doing the modeling. 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 OMNeT++ we'll say *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 first, 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 strategy 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 in order to maintain causality, that is, to ensure that no 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 happens rarely 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 Simple modules in OMNeT++

In OMNeT++, events occur inside simple modules. Simple modules encapsulate C++ code that generates events and reacts to events, in other words, implements the behaviour of the model.

The user creates simple module types by subclassing the `cSimpleModule` class, which is part of the OMNeT++ class library. `cSimpleModule`, just as `cCompoundModule`, is derived from a common base class, `cModule`.

`cSimpleModule`, although packed with simulation-related functionality, doesn't do anything useful by itself -- you have to redefine some virtual member functions to make it do useful work.

These member functions are the following:

- void `initialize()`
- void `handleMessage(cMessage *msg)`
- void `activity()`
- void `finish()`

In the initialization step, OMNeT++ builds the network: it creates the necessary simple and compound modules and connects them according to the NED definitions. OMNeT++ also calls the `initialize()` functions of all modules.

The `handleMessage()` and `activity()` functions are called during event processing. This means that the user will implement the model's behavior in these functions. `handleMessage()` and `activity()` implement different event processing strategies: for each simple module, the user has to redefine exactly one of these functions.

`handleMessage()` is a method that is called by the simulation kernel when the module receives a message.

`activity()` is a coroutine-based solution which implements the process interaction approach (coroutines are non-preemptive (i.e. cooperative) threads). Generally, it is recommended that you prefer `handleMessage()` to `activity()` -- mainly because `activity()` doesn't scale well. Later in this chapter we'll discuss both methods including their advantages and disadvantages.

Modules written with `activity()` and `handleMessage()` can be freely mixed within a simulation model.

The `finish()` functions are called when the simulation terminates successfully. The most typical use of `finish()` is the recording of statistics collected during simulation.

4.1.4 Events in OMNeT++

OMNeT++ uses messages to represent events. Each event is represented by an instance of the `cMessage` class or one its subclasses; there is no separate event class. Messages are sent from one module to another -- this 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:

- the message with **earlier arrival time** is executed first. If arrival times are equal,
- the one with **smaller priority value** is executed first. If priorities are the same,
- the one **scheduled or sent earlier** is executed first.

Priority is a user-assigned integer attribute of messages.

4.1.5 Simulation time

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

Simulation time in OMNeT++ 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 between 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)	+/- *9.22*s
-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)	+/- *2.9227e8* years
0	1s	+/- *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.

The SimTime class performs additions and substractions as 64-bit integer operations. Integer overflows are checked, and will cause the simulation to stop with an error message. Other operations (multiplication, division, etc) are performed in double, then converted back to integer.

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 to convert. To reduce the need for dbl(), several functions and methods have overloaded variants that directly accept SimTime, for example fabs(), fmod(), ceil(), floor(), uniform(), exponential(), and normal().

> **NOTE**
> Converting a SimTime to double may lose precision, because double only has a 52-bit mantissa.

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 int64 value; getScaleExp() which returns the global scale exponent; and getMaxTime which returns the maximum simulation time that can be represented at the current scale exponent.

Compatibility

Earlier versions of OMNeT++ used double for simulation time. To facilitate porting existing models to OMNeT++ 4.0 or later, OMNeT++ can be compiled to use double for simtime_t. To enable this mode, define the USE_DOUBLE_SIMTIME preprocessor macro during compiling OMNeT++ and the simulation models.

There are several macros that can be used in simulation models to make them compile with both double and SimTime simulation time: SIMTIME_STR() converts simulation time to a const char * (can be used in printf argument lists); SIMTIME_DBL(t) converts simulation time to double; SIMTIME_RAW(t) returns the underlying int64 or double; STR_SIMTIME(s) converts string to simulation time; and SIMTIME_TTOA(buf,t) converts simulation time to string, and places the result into the given buffer. MAXTIME is also defined correctly for both simtime_t types.

> **NOTE**
> Why did OMNeT++ switch to int64-based simulation time? double's mantissa is only 52 bits long, and this 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 != x+(y+z)*, see ~[Goldberg91what]) which meant that two double simulation times could not be reliably compared for equality.

4.1.6 FES implementation

The implementation of the FES is a crucial factor in the performance of a discrete event simulator. In OMNeT++, the FES is implemented with *binary heap*, the most widely used data structure for this purpose. Heap is also the best algorithm we know, although exotic data structures like *skiplist* may perform better than heap in some cases. In case you're interested, the FES implementation is in the cMessageHeap class, but as a simulation programmer you won't ever need to care about that.

4.2 Defining simple module types

4.2.1 Overview

As mentioned before [4.1.3], 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 OMNeT++ 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.

> [For completeness, there is also a `Define_Module_Like()` macro, but its use is discouraged and might even be removed in future OMNeT++ releases.]

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>

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

// register module class with `\opp`
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;
}
```

4.2.2 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:

- `cSimpleModule()` -- one without arguments
- `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 OMNeT++ 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.

4.2.3 Constructor and destructor vs initialize() and finish()

The `initialize()` and `finish()` methods will be discussed in a later section in detail, but because their apparent similarity to the constructor and the destructor is prone to cause some confusion, we'll briefly cover them here.

The constructor gets called when the module is created, as part of the model setup process. At that time, everything is just being built, so there isn't a lot things one can do from the constructor. In contrast, `initialize()` gets 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, the following conventions exist for these four methods:

- Constructor:

Set pointer members of the module class to `NULL`; 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.

OMNeT++ prints the list of unreleased objects at the end of the simulation. Simulation models that dump

"undisposed object ..." messages need to get their module destructors 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 OMNeT++ 4.0. Automatic garbage collection cannot be implemented reliably, due to the limitations of the C++ language.

4.2.4 An example

The following code is a bit longer but actually useful simple module implementation. It demonstrates several of the above concepts, plus some others which will be explained in later sections:

- constructor, initialize and destructor conventions
- using messages for timers
- accessing module parameters
- recording statistics at the end of the simulation
- documenting the programmer's assumptions using ASSERT()

```
// file: FFGenerator.h

#include <omnetpp.h>

/**
 * Generates messages or jobs; see NED file for more info.
 */
class FFGenerator : public cSimpleModule
{
  private:
    cMessage *sendMessageEvent;
    long numSent;

  public:
    FFGenerator();
    virtual ~FFGenerator();

  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
};
```

```
// file: FFGenerator.cc

#include "FFGenerator.cc"

// register module class with `\opp`
Define_Module(FFGenerator);

FFGenerator::FFGenerator()
{
    sendMessageEvent = NULL;
}

void FFGenerator::initialize()
{
    numSent = 0;
    sendMessageEvent = new cMessage("sendMessageEvent");
    scheduleAt(0.0, sendMessageEvent);
}

void FFGenerator::handleMessage(cMessage *msg)
{
    ASSERT(msg==sendMessageEvent);
```

```
    cMessage *m = new cMessage("packet");
    m->setBitLength(par("msgLength"));
    send(m, "out");
    numSent++;

    double deltaT = (double)par("sendIaTime");
    scheduleAt(simTime()+deltaT, sendMessageEvent);
}

void FFGenerator::finish()
{
    recordScalar("packets sent", numSent);
}

FFGenerator::~FFGenerator()
{
    cancelAndDelete(sendMessageEvent);
}
```

The corresponding NED declaration:

```
// file: FFGenerator.ned
simple FFGenerator
{
    parameters:
        volatile double sendIaTime;
    gates:
        output out;
}
```

4.2.5 Using 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 Tkenv, or start another run in Cmdenv. This may produce surprising results. Second, they prevent you from running your simulation in parallel. When using parallel simulation, each partition of your model (may) run in a separate process, having its own copy of the 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 . Examples of such modules are the `Blackboard` in the *Mobility Framework*, and `InterfaceTable` and `RoutingTable` in the *INET Framework*.

4.3 Adding functionality to cSimpleModule

This section discusses `cSimpleModule`'s four previously mentioned member functions, intended to be redefined by the user: `initialize()`, `handleMessage()`, `activity()` and `finish()`, plus a fifth, less frequently used one, `handleParameterChange`.

4.3.1 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 OMNeT++ 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()`

You cannot use the `receive()` family and `wait()` functions 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

You can initialize these variables from the `initialize()` function. The constructor is not a very good place for this purpose, because it is called in the network setup phase when the model is still under construction, so a lot of information you may want to use is not yet available.

Another task you have to do 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()`:

- 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.
- For modules which maintain little or no state information, such as packet sinks, `handleMessage()` is more convenient to program.
- 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()`, or the result is code which is better suited for `handleMessage()` (see rule of thumb below). Most communication protocols are like this.

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 pseoudocode:

```
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 to `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.

OMNeT++'s FSM support is described in the next section.

4.3.2 activity()

Process-style description

With `activity()`, you can code the simple module much like you would code an operating system process or a thread. You can wait for an incoming message (event) at any point of the code, you can 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 you can use 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.

There is one scenario where `activity()`'s process-style description is convenient: when the process has many states but transitions are very limited, ie. 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's 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 you have to handle several connections simultaneously, you may dynamically create them as instances of the simple module above. Dynamic module creation will be discussed later.

There are situations when you certainly *do not want* to use `activity()`. If your `activity()` function contains no `wait()` and it has only one `receive()` call at the top of an infinite loop, there's no point in using `activity()` and the code should be written with `handleMessage()`. The body of the infinite loop would then become the body to `handleMessage()`, state variables inside `activity()` would become data members in the module class, and you'd initialize them 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 a sort of threads which are scheduled non-preemptively (this is also called cooperative multitasking). From one coroutine you can switch to another coroutine by a `transferTo(otherCoroutine)` call. Then this coroutine is suspended and *otherCoroutine* will run. Later, when *otherCoroutine* does a `transferTo(firstCoroutine)` call, 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 must have its own processor stack, and `transferTo()` involves a switch from one processor stack to another.

Coroutines are at the heart of OMNeT++, 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 the 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 OMNeT++:

```
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 OMNeT++ at the beginning of the simulation, even before the `initialize()` functions are called.

Coroutine stack size

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

16 or 32 kbytes is usually a good choice, but you may need more if the module uses recursive functions or has local variables, which occupy a lot of stack space. OMNeT++ has a built-in mechanism that will usually detect if the module stack is too small and overflows. OMNeT++ can also tell you how much stack space a module actually uses, so you can find out if you overestimated the stack needs.

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 you have several thousands or ten thousands of simple modules;
- run-time overhead: switching between coroutines is somewhat slower than a simple function call
- does not enforce a good programming style: using `activity()` tends to lead to unreliable, spaghetti code

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 OMNeT++. PARSEC, being ``just'' a programming language, it has a more elegant syntax but far fewer features than OMNeT++.
- Many Java-based simulation libraries are based on Java threads.

4.3.3 initialize() and finish()

Purpose

`initialize()` -- to provide place for any user setup code

`finish()` -- to provide place where the user can record statistics after the simulation has completed

When and how they are called

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 (i.e. not with a runtime error). The calling order is the reverse of the order of `initialize()`: first submodules, then the encompassing compound module. (The bottom line is that at the moment there is no ``official'' possibility to redefine `initialize()` and `finish()` for compound modules; the unofficial way is to write into the nedtool-generated C++ code. Future versions of OMNeT++ will support adding these functions to compound modules.)

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
}
```

initialize() vs. constructor

Usually you should not put simulation-related code into the simple module constructor. This is because modules often need to investigate their surroundings (maybe the whole network) at the beginning of the simulation and save the collected info into internal tables. Code like that cannot be placed into the constructor since the network is still being set up when the constructor is called.

finish() vs. destructor

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, when 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:

```
void initialize(int stage);
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.

> [Note `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.]

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

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 solved in several 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.

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).

4.3.4 handleParameterChange()

The `handleParameterChange()` method was added in OMNeT++ 3.2, and it gets called by the simulation kernel when a module parameter changes. The method signature is the following:

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

The user can redefine this method to let the module react to runtime parameter changes. A typical use is to re-read the changed parameter, and update the module state if needed. For example, if a timeout value changes, one can restart or modify running timers.

The primary motivation for this functionality was to facilitate the implementation of *scenario manager* modules which can be programmed to change parameters at certain simulation times. Such modules can be very convenient in studies involving transient behaviour.

The following example shows a queue module, which supports runtime change of its `serviceTime` parameter:

```
void Queue::handleParameterChange(const char *parname)
{
    if (strcmp(parname, "serviceTime")==0)
    {
        // queue service time parameter changed, re-read it
        serviceTime = par("serviceTime");

        // if there any job being serviced, modify its service time
        if (endServiceMsg->isScheduled())
        {
            cancelEvent(endServiceMsg);
            scheduleAt(simTime()+serviceTime, endServiceMsg);
        }
    }
}
```

4.3.5 Reusing module code via subclassing

It is often needed to have several variants of a simple module. A good design strategy is to create a simple module class with the common functionality, then subclass from it to create the specific simple module types.

An example:

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

Define_Module(ModifiedTransportProtocol);

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

4.4 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 `cModule`:

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

cPar's value can be read with methods that correspond to the parameter's NED type: `boolValue()`, `longValue()`, `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").longValue();
double processingDelay = par("processingDelay"); // using operator double()
```

Note that cPar has two methods for returning 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. In that case you have to clarify by adding either an explicit cast (`(double)par("foo")` or `(long)par("foo")`) or use the `doubleValue()` or `longValue()` methods.

4.4.1 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 get 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 gets 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 spared 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);
    ...
}
```

4.4.2 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 let a module get notified about parameter changes, override its `handleParameterChange()` method, see .

4.4.3 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 gets 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 for users.

4.4.4 Emulating parameter arrays

As of version 4.0, OMNeT++ 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 `doubles` (`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 Accessing gates and connections

4.5.1 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. The `hasGate()` method can be used then. 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.

```
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 `NULL` 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 *the ID of gate k in a gate vector equals the ID of gate 0 plus the index*. 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()`, which also works if the gate 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**
> Earlier versions of OMNeT++ 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. OMNeT++ 4.0 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 = getBaseId("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, output or inout (it returns `cGate::INOUT`, `cGate::INPUT`, or `cGate::OUTPUT`).

Clearly, the above solution can be quite hairy. 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 OMNeT++ 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 (`NULL`s). This is no longer the case with OMNeT++ 4.0 and later versions. Moreover, the `gate()` method now throws an error when called with an invalid ID, and not just returns `NULL`.

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 OMNeT++ 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. 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.) The `isVector()`, `getIndex()`, `getVectorSize()`, `getId()` method names speak for themselves. `size()` is an alias to `getVectorSize()`.

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.

4.5.2 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 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* / *next gate* pointers with various gate types is illustrated on figure below.

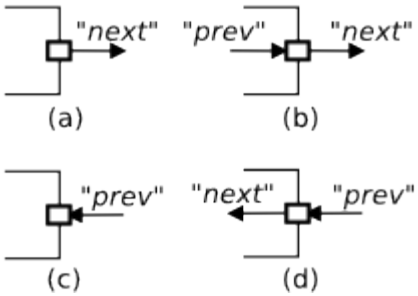


Figure: (a) simple module output gate, (b) compound module output gate, (c) simple module input gate, (d) compound module input gate

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

The `isConnectedOutside()` and `isConnectedInside()` methods return whether a gate is connected on the outside or on the inside. They examine either the *from* or the *to* pointer, depending on the gate type (input or output). Again, see figure below 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;
```

4.5.3 The connection's channel

TODO

4.6 Sending and receiving messages

On an abstract level, an OMNeT++ 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 -- everything else is supposed to facilitate this task, and collect statistics about what was going on.

Messages in OMNeT++ are instances of the `cMessage` class or one of its subclasses. Message objects are created using the C++ `new` operator and destroyed using the `delete` operator when they are no longer needed. During their lifetimes, messages travel between modules via gates and connections (or are sent directly, bypassing the connections), or they are scheduled by and delivered to modules, representing internal events of that module.

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. Message objects can be given descriptive names (a `const char *` string) that often helps in debugging the simulation. The message name string can be specified in the constructor, so it should not surprise you if you see something like `new cMessage("token")` in the examples below.

4.6.1 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 through the gate array.

Examples:

```
send(msg, "outGate");
send(msg, "outGates", i); // send via outGates[i]
```

The following code example creates and sends messages every 5 simulated seconds:

```
int outGateId = findGate("outGate");
while(true)
{
  send(new cMessage("job"), outGateId);
  wait(5);
}
```

4.6.2 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* or simply *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 channel is called the *transmission channel*.

Transmitting a packet

The first packet can be simply send out on the output gate. However, subsequent packets may only be sent when the transmission channel is free, that is, it has finished transmitting earlier packets.

You can get a pointer to the transmission channel by calling the `getDatarateChannel()` 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, a timer (self-message) needs to be scheduled, and the packet should be stored until then, for example in a queue.

The output gate also has `isBusy()` and `getTransmissionFinishTime()` methods, which are basically shortcuts to `getDatarateChannel()->isBusy()` and `getDatarateChannel()->getTransmissionFinishTime()`. When performance is important, it is recommended to obtain a pointer to the

transmission channel once, and then call `isBusy()` and `getTransmissionFinishTime()` on the cached channel pointer.

An incomplete code example to illustrate the above process:

```
simtime_t txfinishTime = gate("out")->getTransmissionFinishTime();
if (txfinishTime <= simTime())
    send(pkt, "out");
else
    scheduleAt(txFinishTime, timerMsg); // also: remember pkt,
    // and ensure that when timerMsg expires, it will get sent out
```

Receiving a 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 `setDeliverOnReceptionStart()` method:

```
gate("in")->setDeliverOnReceptionStart(true);
```

This method may only be called on simple module input gates, and it instructs the simulation kernel to give arriving packets to the receiver module when reception *begins* not ends, that is, on arrival of the first bit of the message. `getDeliverOnReceptionStart()` only needs to be called once, so it is usually done in the `initialize()` method of the module.

When a packet gets 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.

4.6.3 Delay, data rate, bit error rate, packet error rate

Connections can be assigned three parameters, which facilitate the modeling of communication networks, but can be useful for other models too:

- propagation delay (sec)
- bit error rate (errors/bit)
- data rate (bits/sec)

Each of these parameters is optional. One can specify link parameters individually for each connection, or define link types (also called *channel types*) once and use them throughout the whole model.

The *propagation delay* is the amount of time the arrival of the message is delayed by when it travels through the channel. Propagation delay is specified in seconds.

The *bit error rate* has influence on the transmission of messages through the channel. The bit error rate (*ber*) is the probability that a bit is incorrectly transmitted. Thus, the probability that a message of *n* bits length is transferred without bit errors is:

$P_{no\ bit\ error} = (1 - ber)^{length}$

The message has an error flag which is set in case of transmission errors.

The *data rate* is specified in bits/second, and it is used for transmission delay calculation. The sending time of the message normally corresponds to the transmission of the first bit, and the arrival time of the message corresponds to the reception of the last bit (Fig. below).

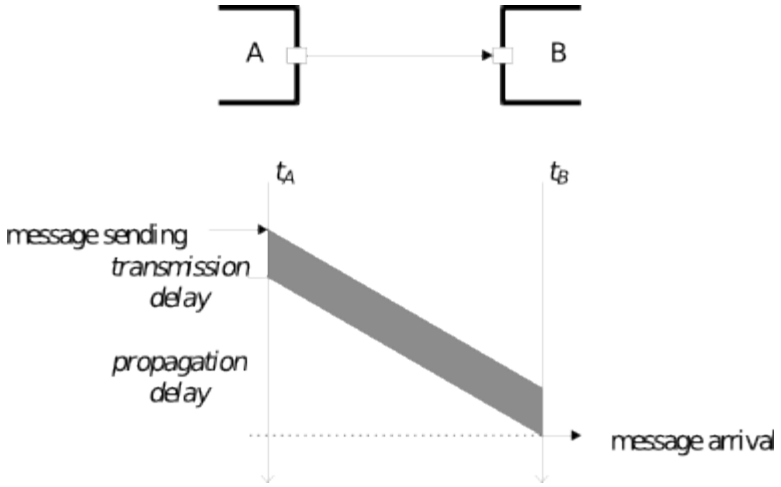


Figure: Message transmission

The above model may not be suitable to model all protocols. In Token Ring and FDDI, stations start to repeat bits before the whole frame arrives; in other words, frames ``flow through'' the stations, being delayed only a few bits. In such cases, the data rate modeling feature of OMNeT++ cannot be used.

If a message travels along a path, passing through successive links and compound modules, the model behaves as if each module waited until the last bit of the message arrives and only started its transmission afterwards. (Fig. below).

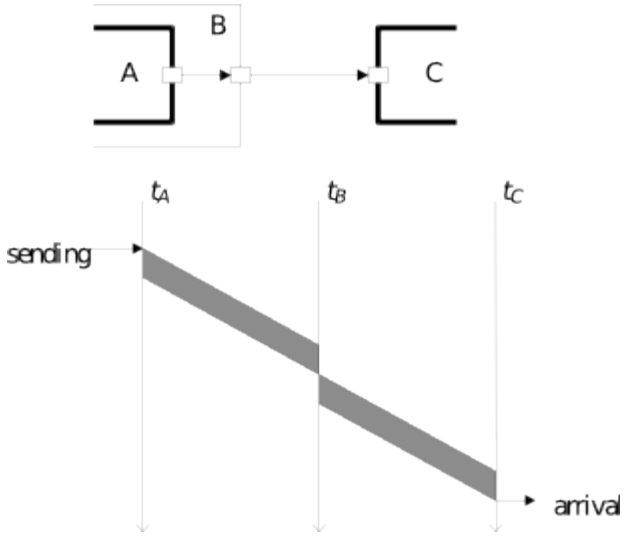


Figure: Message sending over multiple channels

Since the above effect is usually not the desired one, typically you will want to assign data rate to only one connection in the path.

Multiple transmissions on links

If a data rate is specified for a connection, a message will have a certain nonzero transmission time, depending on the length of the connection. This implies that a message that is passsing through an output gate, ``reserves'' the gate for a given period (``it is being transmitted'').

Figure: Connection with a data rate

While a message is under transmission, other messages have to wait until the transmission is completed. The module sends another message while the gate is busy, a runtime error will be thrown.

The OMNeT++ class library provides functions to check whether a certain output gate is transmitting and find out when when it finishes transmission.

If the connection with a data rate is not directly connected to the simple module's output gate but is the second one in the path, you have to check the second gate's busy condition.

> **NOTE**
> In OMNeT++ versions prior to 4.0, sending on a busy gate was permitted, and messages got implicitly queued up. The behaviour of the simulation kernel was changed because in practice, sending on a busy gate was more often result of a programming error than calculated behaviour.

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 gets 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, bit rate, bit error rate) will only affect messages *sent* after the change. Messages already under way 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 approach of some other simulators

Note that some simulators (e.g. OPNET) assign *packet queues* to input gates (ports), and messages sent are buffered at the destination module (or the remote end of the link) until they are received by the destination module. With that approach, events and messages are separate entities, that is, a *send* operation includes placing the message in the packet queue *and* scheduling an event, which signals the arrival of the packet. In some implementations, output gates also have packet queues where packets will be buffered until the channel is ready (available for transmission).

OMNeT++ gates don't have associated queues. The place where sent but not yet received messages are buffered in the FES. OMNeT++'s approach is potentially faster than the solution mentioned above because it doesn't have the enqueue/dequeue overhead and also spares an event creation. The drawback is, that changes to channel parameters do not take effect immediately.

In OMNeT++ one can implement *point-to-point transmitter* modules with packet queues if needed. For example, the INET Framework follows this approach.

Connection attributes (propagation delay, transmission data rate, bit error rate) are represented by the channel object, which is available via the source gate of the connection.

```
cChannel *chan = outgate->getChannel();
```

`cChannel` is a small base class. All interesting attributes are part of its subclass `cDatarateChannel`, so you have to cast the pointer before getting to the delay, error and data rate values.

```
cDatarateChannel *chan = check_and_cast<cDatarateChannel *>(outgate->getChannel());
```

```
double d = chan->getDelay();
double e = chan->getBitErrorRate();
double r = chan->getDatarate();
```

You can also change the channel attributes with the corresponding `setXXX()` functions. Note, however, that (as it was explained in section) changes will not affect messages already sent, even if they have not begun transmission yet.

Channel transmission state

The `isBusy()` member function returns whether the gate is currently transmitting, and if so, the `getTransmissionFinishTime()` member function returns the simulation time when the gate is going to finish transmitting. (If the gate in not currently transmitting, `getTransmissionFinishTime()` returns the simulation time when it finished its last transmission.)

An example:

```
cMessage *packet = new cMessage("DATA");
packet->setByteLength(1024);  // 1K

if (gate("TxGate")->isBusy()) // if gate is busy, wait until it
{                             // becomes free
  wait( gate("TxGate")->getTransmissionFinishTime() - simTime());
}
send( packet, "TxGate");
```

If the connection with a data rate is not directly connected to the simple module's output gate but is the second one in the path, you have to check the second gate's busy condition. You could use the following code:

```
if (gate("out")->getNextGate()->isBusy())
  //...
```

Note that if data rates change during the simulation, the changes will affect only the messages that are *sent* after the change.

4.6.4 Broadcasts and retransmissions

When you implement 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. The solution in such cases is duplicating the message.

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 spare looking up the gate by name for each send operation. 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);
```

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, whenever it comes to (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;
```

Why?

A message is like any real world object -- it cannot be at two places at the same time. Once you've sent it, the message object 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 arrived 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 you actually own the message you are about to send. If the message is with another module, it is currently scheduled or in a queue etc., you'll get a runtime error: *not owner of message*.

> [The feature does not increase runtime overhead significantly, because it uses the object ownership management (described in Section [6.11]); it merely checks that the owner of the message is the module that wants to send it.]

4.6.5 Delayed sending

It is often needed to model a delay (processing time, etc.) immediately followed by message sending. In OMNeT++, it is possible to implement it like this:

```
wait(someDelay);
send(msg, "outgate");
```

If the module needs to react to messages that arrive during the delay, `wait()` cannot be used and the timer mechanism described in Section [4.6.9], ``Self-messages'', would need to be employed.

There is also a more straightforward method than those mentioned above: delayed sending. Delayed sending can

be achieved by using one of these functions:

```
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 effect of the function is the same as if the module had kept the message for the delay interval and sent it afterwards. That is, the sending time of the message will be the current simulation time (time at the `sendDelayed()` call) plus the delay. The delay value must be non-negative.

Example:

```
sendDelayed(msg, 0.005, "outGate");
```

4.6.6 Direct message sending

Sometimes it is necessary or convenient to ignore gates/connections and send a message directly to a remote destination module. The `sendDirect()` function does that:

```
sendDirect(cMessage *msg, double delay, cModule *mod, int gateId)
sendDirect(cMessage *msg, double delay, cModule *mod, const char *gateName, int
index=-1)
sendDirect(cMessage *msg, double delay, cGate *gate)
```

In addition to the message and a delay, it also takes the destination module and gate. The gate should be an *input* gate and should not be connected. In other words, the module needs dedicated gates for receiving via `sendDirect()`. (Note: For leaving a gate unconnected in a compound module, you'll need to specify `connections nocheck:` instead of plain `connections:` in the NED file.)

An example:

```
cModule *destinationModule = getParentModule()->getSubmodule("node2");
double delay = truncnormal(0.005, 0.0001);
sendDirect(new cMessage("packet"), delay, destinationModule, "inputGate");
```

At the destination module, there is no difference between messages received directly and those received over connections.

4.6.7 Receiving messages

With activity() only! The message receiving functions can only be used in the `activity()` function, `handleMessage()` gets received messages in its argument list.

Messages are received using the `receive()` function. `receive()` is a member of `cSimpleModule`.

```
cMessage *msg = receive();
```

The `receive()` function accepts an optional *timeout* parameter. (This is a *delta*, not an absolute simulation time.) If an appropriate message doesn't arrive within the timeout period, the function returns a NULL pointer.

> [Putaside-queue and the functions `receiveOn()`, `receiveNew()`, and `receiveNewOn()` were deprecated in OMNeT++ 2.3 and removed in OMNeT++ 3.0.]

```
simtime_t timeout = 3.0;
cMessage *msg = receive( timeout );

if (msg==NULL)
{
    ...   // handle timeout
}
else
{
    ...  // process message
}
```

4.6.8 The wait() function

With activity() only! The `wait()` function's implementation contains a `receive()` call which cannot be used in `handleMessage()`.

The `wait()` function suspends the execution of the module for a given amount of simulation time (a *delta*).

```
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 a (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 [6]) in addition to the wait interval. Messages that arrive during the wait interval will be accumulated in the queue, so you can process them after the `waitAndEnqueue()` call returned.

```
cQueue queue("queue");
...
waitAndEnqueue(waitTime, &queue);
if (!queue.empty())
{
  // process messages arrived during wait interval
  ...
}
```

4.6.9 Modeling events using self-messages

In most simulation models it is necessary to implement timers, or schedule events that occur at some point in the future. For example, when a packet is sent by a communications protocol model, it has to schedule an event that would occur when a timeout expires, because it will have to resent the packet then. As another example, suppose you want to write a model of a server which processes jobs from a queue. Whenever it begins processing a job, the server model will want to schedule an event to occur when the job finishes processing, so that it can begin processing the next job.

In OMNeT++ 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. Self-messages are used to model events which occur within the module.

Scheduling an event

The module can send a message to itself using the `scheduleAt()` function. `scheduleAt()` accepts an *absolute* simulation time, usually calculated as `simTime()`+*delta*:

```
scheduleAt(absoluteTime, msg);
scheduleAt(simtime()+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.

As an example, here's how you could implement your own `wait()` function in an `activity()` simple module, if the simulation kernel didn't provide it already:

```
cMessage *msg = new cMessage();
scheduleAt(simtime()+waitTime, msg);
cMessage *recvd = receive();
if (recvd!=msg)
   // hmm, some other event occurred meanwhile: error!
...
```

You can determine if a message is currently in the FES by calling its `isScheduled()` member:

```
if (msg->isScheduled())
  // currently scheduled
else
  // not scheduled
```

Re-scheduling an event

If you want to reschedule an event which is currently scheduled to a different simulation time, first you have to cancel it using `cancelEvent()`.

Cancelling an event

Scheduled self-messages can be cancelled (removed from the FES). This is particularly useful because self-messages are often used to model timers.

```
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 the next `scheduleAt()` calls. `cancelEvent()` gives an error if the message is not in the FES.

Implementing timers

The following example shows how to implement timers:

```
cMessage *timeoutEvent = new cMessage("timeout");

scheduleAt(simTime()+10.0, timeoutEvent);
//...
```

```
cMessage *msg = receive();
if (msg == timeoutEvent)
{
  // timeout expired
}
else
{
  // other message has arrived, timer can be cancelled now:
  delete cancelEvent(timeoutEvent);
}
```

4.7 Stopping the simulation

4.7.1 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).

4.7.2 Raising errors

If your simulation encounters an error condition, you can throw a `cRuntimeError` exception to terminate the simulation with an error message (and in case of Cmdenv, a nonzero exit code). The `cRuntimeError` class has a constructor whose argument list is similar to `printf()`:

```
if (windowSize<1)
    throw cRuntimeError("Invalid window size %d; must be >=1", windowSize);
```

Do not include a newline (``\n'') or punctuation (period or exclamation mark) in the error text; it will be added by OMNeT++.

You can achieve the same effect by calling the `error()` method of `cModule`.

```
if (windowSize<1)
    error("Invalid window size %d; must be >=1", windowSize);
```

Of course, the `error()` method can only be used when a module pointer is available.

4.8 Finite State Machines in OMNeT++

Overview

Finite State Machines (FSMs) can make life with `handleMessage()` easier. OMNeT++ provides a class and a set of macros to build FSMs. OMNeT++'s FSMs work very much like OPNET's or SDL's.

The key points are:

- There are two kinds of states: *transient* and *steady*. At 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 OMNeT++). State changes (transitions) must be put into the exit code.

OMNeT++'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 would become 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()`, where you have 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,\textit{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 `ev`, 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, you have to `#define FSM_DEBUG` before including `omnetpp.h`.

```
#define FSM_DEBUG    // enables debug output from FSMs
#include <omnetpp.h>
```

The actual logging is done via the `FSM_Print()` macro. It is currently defined as follows, but you can change the output format by undefining `FSM_Print()` after including `omnetpp.ini` and providing a new definition instead.

```
#define FSM_Print(fsm,exiting)
  (ev << "FSM " << (fsm).getName()
      << ((exiting) ? ": leaving state " : ": entering state ")
      << (fsm).getStateName() << endl)
```

Implementation

The `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. (The actual code is rather scary, but if you're dying to see it, it is in `cfsm.h`.)

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 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>

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;
      }
    }
}
```

4.9 Walking the module hierarchy

Module vectors

If a module is part of a module vector, the `getIndex()` and `size()` 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->size() << "].\n";
```

Module IDs

Each module in the network has a unique ID that is returned by the `getId()` member function. The module ID is used internally by the simulation kernel to identify modules.

```
int myModuleId = getId();
```

If you know the module ID, you can ask the simulation object (a global variable) to get back the module pointer:

```
int id = 100;
cModule *mod = simulation.getModule( id );
```

Module IDs are guaranteed to be unique, even when modules are created and destroyed dynamically. That is, an ID which once belonged to a module which was deleted is never issued to another module later.

Walking up and down the module hierarchy

The surrounding compound 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+index if the submodule is in a module vector). The first one returns the numeric module ID of the submodule, and the latter returns the module pointer. If the submodule is not found, they return -1 or NULL, respectively.

```
int submodID = compoundmod->findSubmodule("child",5);
cModule *submod = compoundmod->getSubmodule("child",5);
```

The `getModuleByRelativePath()` member function can be used to find a submodule nested deeper than one level below. For example,

```
compoundmod->getModuleByRelativePath("child[5].grandchild");
```

would give the same results as

```
compoundmod->getSubmodule("child",5)->getSubmodule("grandchild");
```

(Provided that `child[5]` does exist, because otherwise the second version would crash with an access violation because of the NULL pointer dereference.)

The `cSimulation`::`getModuleByPath()` function is similar to `cModule`'s `moduleByRelativePath()` function, and it starts the search at the top-level module.

Iterating over submodules

To access all modules within a compound module, use `cSubModIterator`.

For example:

```
for (cSubModIterator iter(*getParentModule()); !iter.end(); iter++)
{
  ev << iter()->getFullName();
}
```

(`iter()` is pointer to the current module the iterator is at.)

The above method can also be used to iterate along a module vector, since the `getName()` function returns the same for all modules:

```
for (cSubModIterator iter(*getParentModule()); !iter.end(); iter++)
{
  if (iter()->isName(getName())) // if iter() is in the same
                                 // vector as this module
  {
    int itsIndex = iter()->getIndex();
    // do something to it
  }
}
```

Walking along links

To determine the module at the other end of a connection, use `cGate`'s `getPreviousGate()`, `getNextGate()` and `getOwnerModule()` functions. For example:

```
cModule *neighbour = gate("out")->getNextGate()->getOwnerModule();
```

For input gates, you would use `getPreviousGate()` instead of `getNextGate()`.

4.10 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 [4.9].) 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 *calleeModule = getParentModule()->getSubmodule("callee");
Callee *callee = check_and_cast<Callee *>(calleeModule);
callee->doSomething();
```

The `check_and_cast<>()` template function on the second line is part of OMNeT++. It does a standard C++ `dynamic_cast`, and checks the result: if it is NULL, `check_and_cast` raises an OMNeT++ error. Using `check_and_cast` saves you from writing error checking code: if `calleeModule` from the first line is NULL because the submodule named `"callee"` was not found, or if that module is actually not of type `Callee`, an error gets thrown from `check_and_cast`.

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 to several internal mechanisms to work correctly. (One such mechanism is ownership handling.) Second, the Tkenv simulation GUI can animate method calls, but to be able to do that, it has to know about them.

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 call. `Enter_Method()` expects a `printf()`-like argument list -- the resulting string will be displayed during animation.

```
void Callee::doSomething()
{
    Enter_Method("doSomething()");
    ...
}
```

4.11 Dynamic module creation

4.11.1 When do you need dynamic module creation

In some situations you need to dynamically create and maybe destroy modules. For example, when simulating a mobile network, you may create a new module whenever a new user enters the simulated area, and dispose of them when they leave the area.

As another example, when implementing a server or a transport protocol, it might be convenient to dynamically create modules to serve new connections, and dispose of them when the connection is closed. (You would write a manager module that receives connection requests and creates a module for each connection. The Dyna example simulation does something like this.)

Both simple and compound modules can be created dynamically. If you create a compound module, all its submodules will be created recursively.

It is often convenient to use direct message sending with dynamically created modules.

Once created and started, dynamic modules aren't any different from ``static'' modules; for example, one could also delete static modules during simulation (though it is rarely useful.)

4.11.2 Overview

To understand how dynamic module creation works, you have to know a bit about how normally OMNeT++ 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 function which can instantiate the module class (this function basically only consists of a `return new module-class(...)` statement).

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 compiled NED code.)

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 OMNeT++, 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.

4.11.3 Creating modules

The first step, finding the factory object:

```
cModuleType *moduleType = cModuleType::get("WirelessNode");
```

Simplified form

`cModuleType` has a `createScheduleInit(const char *name, cModule *parentmod)` convenience function to get a module up and running in one step.

```
mod = modtype->createScheduleInit("node",this);
```

It does `create()` + `buildInside()` + `scheduleStart(now)` + `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.

Expanded form

If the previous simple form cannot be used. There are 5 steps:

□□□ find factory object
□□□ create module

- set up parameters and gate sizes (if needed)
- call function that builds out submodules and finalizes the module
- call function that creates activation message(s) for the new simple getModule(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("WirelessNode");

// create (possibly compound) module and build its submodules (if any)
cModule *module = moduleType->create("node", this);
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("WirelessNode");
cModule *module = moduleType->create("node", this);

// set up parameters and gate sizes before we set up its submodules
module->par("address") = ++lastAddress;
module->setGateSize("in", 3);
module->setGateSize("out", 3);

// create internals, and schedule it
module->buildInside();
module->scheduleStart(simTime());
```

4.11.4 Deleting modules

To delete a module dynamically:

```
module->deleteModule();
```

If the module was a compound module, this involves recursively destroying all its submodules. A simple module can also delete itself; in this case, the `deleteModule()` call does not return to the caller.

Currently, you cannot safely delete a compound module from a simple module in it; you must delegate the job to a module outside the compound module.

4.11.5 Module deletion and finish()

When you delete a module *during simulation*, its `finish()` function is not called automatically (`deleteModule()` doesn't do it.) How the module was created doesn't play any role here: `finish()` gets called for *all* modules -- at the end of the simulation. If a module doesn't live that long, `finish()` is not invoked, but you can still manually invoke it.

You can use the `callFinish()` function to arrange `finish()` to be called. It is usually not a good idea to invoke `finish()` directly. If you're deleting a compound module, `callFinish()` will recursively invoke `finish()` for all submodules, and if you're deleting a simple module from another module, `callFinish()` will do the context switch for the duration of the call.

[The `finish()` function is even made `protected` in `cSimpleModule`, in order to discourage its invocation from other modules.]

Example:

```
mod->callFinish();
mod->deleteModule();
```

4.11.6 Creating connections

Connections can be created using `cGate`'s `connectTo()` method.

[The earlier `connect()` global functions that accepted two gates have been deprecated, and may be removed from further OMNeT++ releases.]

`connectTo()` should be invoked on the source gate of the connection, and expects the destination gate pointer as an argument:

```
srcGate->connectTo(destGate);
```

The *source* and *destination* words correspond to the direction of the arrow in NED files.

As an example, we create two modules and connect 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"));
```

`connectTo()` also accepts a channel object as an additional, optional argument. Channels are subclassed from `cChannel`. Almost always you'll want use an instance of `cDataRateChannel` as channel -- this is the one that supports delay, bit error rate and data rate. The channel object will be owned by the source gate of the connection, and you cannot reuse the same channel object with several connections.

`cDataRateChannel` has `setDelay()`, `setBitErrorRate()` and `setDatarate()` methods to set up the channel attributes.

An example that sets up a channel with a delay:

```
cDataRateChannel *channel = new cDataRateChannel("channel");
channel->setDelay(0.001);

a->gate("out")->connectTo(b->gate("in"), channel); // a,b are modules
```

4.11.7 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();
```

---

5 Messages

5.1 Messages and packets

5.1.1 The cMessage class

`cMessage` is a central class in OMNeT++. Objects of `cMessage` and subclasses may model a number of things: events; messages; packets, frames, cells, bits or signals travelling in a network; entities travelling in a system and so on.

Attributes

A `cMessage` object has number of attributes. Some are used by the simulation kernel, others are provided just for the convenience of the simulation programmer. A more-or-less complete list:

- The *name* attribute is a string (`const char *`), which can be freely used by the simulation programmer. The message name appears in many places in Tkenv (for example, in animations), and it is generally very useful to choose a descriptive name. This attribute is inherited from `cOwnedObject` (see section [6.1.1]).
- The *message kind* attribute is supposed to carry some message type information. Zero and positive values can be freely used for any purpose. Negative values are reserved for use by the OMNeT++ simulation library.
- The *length* attribute (understood in bits) is used to compute transmission delay when the message travels through a connection that has an assigned data rate.
- The *bit error flag* attribute is set to true by the simulation kernel with a probability of $1-(1-ber)^{length}$ when the message is sent through a connection that has an assigned bit error rate (*ber*).
- The *priority* attribute is used by the simulation kernel to order messages in the message queue (FES) that have the same arrival time values.
- The *time stamp* attribute is not used by the simulation kernel; you can use it for purposes such as noting the time when the message was enqueued or re-sent.
- Other attributes and data members make simulation programming easier, they will be discussed later: *parameter list*, *encapsulated message*, *control info* and *context pointer*.
- A number of read-only attributes store information about the message's (last) sending/scheduling: *source/destination module and gate*, *sending (scheduling) and arrival time*. They are mostly used by the simulation kernel while the message is in the FES, but the information is still in the message object when a module receives the message.

Basic usage

The `cMessage` constructor accepts several arguments. Most commonly, you would create a message using an *object name* (a `const char *` string) and a *message kind* (`int`):

```
cMessage *msg = new cMessage("MessageName", msgKind);
```

Both arguments are optional and initialize to the null string (`""`) and 0, so the following statements are also valid:

```
cMessage *msg = new cMessage();
cMessage *msg = new cMessage("MessageName");
```

It is a good idea to *always* use message names -- they can be extremely useful when debugging or demonstrating your simulation.

Message kind is usually initialized with a symbolic constant (e.g. an *enum* value) which signals what the message object represents in the simulation (i.e. a data packet, a jam signal, a job, etc.) Please use *positive values or zero* only as message kind -- negative values are reserved for use by the simulation kernel.

The cMessage constructor accepts further arguments too (*length*, *priority*, *bit error flag*), but for readability of the code it is best to set them explicitly via the set...() methods described below. Length and priority are integers, and the bit error flag is boolean.

Once a message has been created, its data members can be changed by the following functions:

```
msg->setKind( kind );
msg->setBitLength( length );
msg->setByteLength( lengthInBytes );
msg->setPriority( priority );
msg->setBitError( err );
msg->setTimestamp();
msg->setTimestamp( simtime );
```

With these functions the user can set the message kind, the message length, the priority, the error flag and the time stamp. The setTimeStamp() function without any argument sets the time stamp to the current simulation time. setByteLength() sets the same length field as setBitLength(), only the parameters gets internally multiplied by 8.

The values can be obtained by the following functions:

```
int k       = msg->getKind();
int p       = msg->getPriority();
int l       = msg->getBitLength();
int lb      = msg->getByteLength();
bool b      = msg->hasBitError();
simtime_t t = msg->getTimestamp();
```

getByteLength() also reads the length field as length(), but the result gets divided by 8 and rounded up.

Duplicating messages

It is often necessary to duplicate a message (for example, sending one and keeping a copy). This can be done in the same way as for any other OMNeT++ object:

```
cMessage *copy = msg->dup();
```

or

```
cMessage *copy = new cMessage( *msg );
```

The two are equivalent. The resulting message is an exact copy of the original, including message parameters (cMsgPar or other object types) and encapsulated messages.

5.1.2 Self-messages

Using a message as self-message

Messages are often used to represent events internal to a module, such as a periodically firing timer on 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, you can call the isSelfMessage() method to determine if it is a self-message; it other words, if it was scheduled with scheduleAt() or was 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();
bool isScheduled();
```

The following methods return the time of creating and scheduling the message as well as its arrival time. While the message is scheduled, arrival time is the time it will be delivered to the module.

```
simtime_t getCreationTime();
simtime_t getSendingTime();
simtime_t getArrivalTime();
```

Context pointer

`cMessage` contains a `void*` pointer which is set/returned by the `setContextPointer()` and `getContextPointer()` functions:

```
void *context =...;
msg->setContextPointer(context);
void *context2 = msg->getContextPointer();
```

It 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.

5.1.3 Modelling packets

Arrival gate and time

The following methods can tell where the message came from and where it arrived (or will arrive if it is currently scheduled or under way.)

```
int getSenderModuleId();
int getSenderGateId();
int getArrivalModuleId();
int getArrivalGateId();
```

The following methods are just convenience functions which build on the ones above.

```
cModule *getSenderModule();
cGate *getSenderGate();
cGate *getArrivalGate();
```

And there are further convenience functions to tell whether the message arrived on a specific gate given with id or name+index.

```
bool arrivedOn(int id);
bool arrivedOn(const char *gname, int gindex=0);
```

The following methods return message creation time and the last sending and arrival times.

```
simtime_t getCreationTime();
simtime_t getSendingTime();
simtime_t getArrivalTime();
```

Control info

One of the main application areas of OMNeT++ 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 `cMessage`.

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'll want to let TCP know at least the source IP address.

This additional information is represented by *control info* objects in OMNeT++. Control info objects have to be subclassed from `cObject` (a small footprint base class with no data members), and attached to the messages representing packets. `cMessage` has the following methods for this purpose:

```
void setControlInfo(cObject *controlInfo);
cObject *getControlInfo();
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.

Identifying the protocol

In OMNeT++ 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) and/or in the message kind value. The PDU type is usually represented as a field inside the message class.

The C++ `dynamic_cast` operator can be used to determine if a message object is of a specific protocol.

```
cMessage *msg = receive();
if (dynamic_cast<IPv6Datagram *>(msg) != NULL)
{
    IPv6Datagram *datagram = (IPv6Datagram *)msg;
    ...
}
```

5.1.4 Encapsulation

Encapsulating packets

It is often necessary to encapsulate a message into another when you're modeling layered protocols of computer networks. Although you can encapsulate messages by adding them to the parameter list, there's a better way.

The `encapsulate()` function encapsulates a message into another one. The length of the message will grow by the length of the encapsulated message. An exception: when the encapsulating (outer) message has zero length, OMNeT++ assumes it is not a real packet but some out-of-band signal, so its length is left at zero.

```
cMessage *userdata = new cMessage("userdata");

userdata->setByteLength(2048);  // 2K
cMessage *tcpseg = new cMessage("tcp");
tcpseg->setByteLength(24);
tcpseg->encapsulate(userdata);
ev << tcpseg->getByteLength() << endl; // --> 2048+24 = 2072
```

A message can only hold one encapsulated message at a time. The second `encapsulate()` call will result in an error. It is also an error if the message to be encapsulated isn't owned by the module.

You can get back the encapsulated message by `decapsulate()`:

```
cMessage *userdata = tcpseg->decapsulate();
```

`decapsulate()` will decrease the length of the message accordingly, except if it was zero. If the length would become negative, an error occurs.

The `getEncapsulatedMsg()` function returns a pointer to the encapsulated message, or `NULL` if no message was encapsulated.

Reference counting

Since the 3.2 release, OMNeT++ implements reference counting of encapsulated messages, meaning that if you `dup()` a message that contains an encapsulated message, then the encapsulated message will not be duplicated, only a reference count incremented. Duplication of the encapsulated message is deferred until `decapsulate()` actually gets called. If the outer message gets deleted without its `decapsulate()` method ever being called, then the reference count of the encapsulated message simply gets decremented. The encapsulated message 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 get 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 message that is encapsulated into another!** That is, `getEncapsulatedMsg()` 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 message may be shared between several messages, and any change would affect those other messages as well.

Encapsulating several messages

The `cMessage` class doesn't directly support adding more than one messages to a message object, but you can subclass `cMessage` and add the necessary functionality. (It is recommended that you use the message definition syntax [5.2] and customized messages [5.2.6] to be described later on in this chapter -- it can spare you some work.)

You can store the messages in a fixed-size or a dynamically allocated array, or you can use STL classes like `std::vector` or `std::list`. There is one additional ``trick'' that you might not expect: your message class has to **take ownership** of the inserted messages, and **release** them when they are removed from the message. These are done via the `take()` and `drop()` methods. Let us see an example which assumes you have added to the class an `std::list` member called `messages` that stores message pointers:

```
void MessageBundleMessage::insertMessage(cMessage *msg)
{
    take(msg);  // take ownership
    messages.push_back(msg);  // store pointer
}

void MessageBundleMessage::removeMessage(cMessage *msg)
{
    messages.remove(msg);  // remove pointer
    drop(msg);  // release ownership
}
```

You will also have to provide an `operator=()` method to make sure your message objects can be copied and

duplicated properly -- this is something often needed in simulations (think of broadcasts and retransmissions!). Section [6.10] contains more info about the things you need to take care of when deriving new classes.

5.1.5 Attaching parameters and objects

If you want to add parameters or objects to a message, the preferred way to do that is via message definitions, described in chapter [5.2].

Attaching objects

The `cMessage` class has an internal `cArray` object which can carry objects. Only objects that are derived from `cOwnedObject` (most OMNeT++ classes are so) can be attached. The `addObject()`, `getObject()`, `hasObject()`, `removeObject()` methods use the object name as the key to the array. An example:

```
cLongHistogram *pklenDistr = new cLongHistogram("pklenDistr");
msg->addObject(pklenDistr);
...
if (msg->hasObject("pklenDistr"))
{
   cLongHistogram *pklenDistr =
       (cLongHistogram *) msg->getObject("pklenDistr");
   ...
}
```

You should take care that names of the attached objects do not clash with each other or with `cMsgPar` parameter names (see next section). If you do not attach anything to the message and do not call the `getParList()` function, the internal `cArray` object will not be created. This saves both storage and execution time.

You can attach non-object types (or non-`cOwnedObject` objects) to the message by using `cMsgPar`'s `void*` pointer 'P') type (see later in the description of `cMsgPar`). An example:

```
struct conn_t *conn = new conn_t; // conn_t is a C struct
msg->addPar("conn") = (void *) conn;
msg->par("conn").configPointer(NULL,NULL,sizeof(struct conn_t));
```

Attaching parameters

The preferred way of extending messages with new data fields is to use message definitions (see section [5.2]).

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`s 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 you mistype the name of a field in the C++ code, already the compiler can detect the mistake.

If you still need `cMsgPar`s for some reason, here's a short summary. At the sender side you 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, you 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");
```

5.2 Message definitions

5.2.1 Introduction

In practice, you'll need to add various fields to `cMessage` to make it useful. For example, if you're modelling packets in communication networks, you need to have a way to store protocol header fields in message objects. Since the simulation library is written in C++, the natural way of extending `cMessage` is via subclassing it. However, because for each field you need to write at least three things (a private data member, a getter and a setter method), and the resulting class has to integrate with the simulation framework, writing the necessary C++ code can be a tedious and time-consuming task.

OMNeT++ offers a more convenient way called *message definitions*. Message definitions provide a very compact syntax to describe message contents. C++ code is automatically generated from message definitions, saving you a lot of typing.

A common source of complaint about code generators in general is lost flexibility: if you have a different idea how the generated code should look like, there's little you can do about it. In OMNeT++, however, there's nothing to worry about: you can customize the generated class to any extent you like. Even if you decide to heavily customize the generated class, message definitions still save you a great deal of manual work.

The subclassing approach for adding message parameters was originally suggested by Nimrod Mesika.

The first message class

Let us begin with a simple example. Suppose that you need message objects to carry source and destination addresses as well as a hop count. You could write a `mypacket.msg` file with the following contents:

```
message MyPacket
{
    int srcAddress;
    int destAddress;
    int hops = 32;
};
```

The task of the *message subclassing compiler* is to generate C++ classes you can use from your models as well as ``reflection'' classes that allow Tkenv to inspect these data structures.

If you process `mypacket.msg` with the message subclassing compiler, it will create the following files for you: `mypacket_m.h` and `mypacket_m.cc`. `mypacket_m.h` contains the declaration of the `MyPacket` C++ class, and it should be included into your C++ sources where you need to handle `MyPacket` objects.

The generated `mypacket_m.h` will contain the following class declaration:

```
class MyPacket : public cMessage {
    ...
    virtual int getSrcAddress() const;
    virtual void setSrcAddress(int srcAddress);
    ...
};
```

So in your C++ file, you could use the `MyPacket` class like this:

```
#include "mypacket_m.h"

...
MyPacket *pkt = new MyPacket("pkt");
pkt->setSrcAddress( localAddr );
...
```

The `mypacket_m.cc` file contains implementation of the generated `MyPacket` class, as well as ``reflection'' code that allows you to inspect these data structures in the Tkenv GUI. The `mypacket_m.cc` file should be compiled and linked into your simulation. (If you use the `opp_makemake` tool to generate your makefiles, the latter will be automatically taken care of.)

What is message subclassing *not?*

There might be some confusion around the purpose and concept of message definitions, so it seems to be a good idea to deal with them right here.

It is ***not:***

- *... an attempt to reproduce the functionality of C++ with another syntax.* Do not look for complex C++ types, templates, conditional compilation, etc. Also, it defines *data* only (or rather: an interface to access data) -- not any kind of active behaviour.
- *... a generic class generator.* This is meant for defining message contents, and data structure you put in messages. Defining methods is not supported on purpose. Also, while you can probably (ab)use the syntax to generate classes and structs used internally in simple modules, this is probably not a good idea.

The goal is to define the *interface* (getter/setter methods) of messages rather than their implementations in C++. A simple and straightforward implementation of fields is provided -- if you'd like a different internal representation for some field, you can have it by customizing the class.

There are questions you might ask:

- *Why doesn't it support `std::vector` and other STL classes?* Well, it does. Message definitions focus on the interface (getter/setter methods) of the classes, optionally leaving the implementation to you -- so you can implement fields (dynamic array fields) using `std::vector`. (This aligns with the idea behind STL -- it was designed to be *nuts and bolts* for C++ programs).
- *Why does it support C++ data types and not octets, bytes, bits, etc..?* That would restrict the scope of message definitions to networking, and OMNeT++ wants to support other application areas as well. Furthermore, the set of necessary concepts to be supported is probably not bounded, there would always be new data types to be adopted.
- *Why no embedded classes?* Good question. As it does not conflict with the above principles, it might be added someday.

The following sections describe the message syntax and features in detail.

5.2.2 Declaring enums

An enum (declared with the **enum** keyword) generates a normal C++ enum, plus creates an object which stores text representations of the constants. The latter makes it possible to display symbolic names in Tkenv. An example:

```
enum ProtocolTypes
{
   IP = 1;
   TCP = 2;
};
```

Enum values need to be unique.

5.2.3 Message declarations

Basic use

You can describe messages with the following syntax:

```
message FooPacket
{
    int sourceAddress;
    int destAddress;
    bool hasPayload;
};
```

Processing this description with the message compiler will produce a C++ header file with a generated class, `FooPacket`. `FooPacket` will be a subclass of `cMessage`.

For each field in the above description, the generated class will have a protected data member, a 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. Thus, `FooPacket` will contain the following methods:

```
virtual int getSourceAddress() const;
virtual void setSourceAddress(int sourceAddress);

virtual int getDestAddress() const;
virtual void setDestAddress(int destAddress);

virtual bool getHasPayload() const;
virtual void setHasPayload(bool hasPayload);
```

Note that the methods are all declared **`virtual`** to give you the possibility of overriding them in subclasses.

Two constructors will be generated: one that optionally accepts object name and (for `cMessage` subclasses) message kind, and a copy constructor:

```
FooPacket(const char *name=NULL, int kind=0);
FooPacket(const FooPacket& other);
```

Appropriate assignment operator (`operator=()`) and `dup()` methods will also be generated.

Data types for fields are not limited to **`int`** and **`bool`**. You can use the following primitive types (i.e. primitive types as defined in the C++ language):

- **`bool`**
- **`char`**, **`unsigned char`**
- **`short`**, **`unsigned short`**
- **`int`**, **`unsigned int`**

- **long**, **unsigned long**
- **double**

Field values are initialized to zero.

Initial values

You can initialize field values with the following syntax:

```
message FooPacket
{
    int sourceAddress = 0;
    int destAddress = 0;
    bool hasPayload = false;
};
```

Initialization code will be placed in the constructor of the generated class.

Enum declarations

You can declare that an **int** (or other integral type) field takes values from an enum. The message compiler can than generate code that allows Tkenv display the symbolic value of the field.

Example:

```
message FooPacket
{
    int payloadType enum(PayloadTypes);
};
```

The enum has to be declared separately in the message file.

Fixed-size arrays

You can specify fixed size arrays:

```
message FooPacket
{
    long route[4];
};
```

The generated getter and setter methods will have an extra k argument, the array index:

```
virtual long getRoute(unsigned k) const;
virtual void setRoute(unsigned k, long route);
```

If you call the methods with an index that is out of bounds, an exception will be thrown.

Dynamic arrays

If the array size is not known in advance, you can declare the field to be a dynamic array:

```
message FooPacket
{
    long route[];
};
```

In this case, the generated class will have two extra methods in addition to the getter and setter methods: one for setting the array size, and another one for returning the current array size.

```
virtual long getRoute(unsigned k) const;
virtual void setRoute(unsigned k, long route);
virtual unsigned getRouteArraySize() const;
virtual void setRouteArraySize(unsigned n);
```

The `set...ArraySize()` method internally allocates a new array. Existing values in the array will be preserved (copied over to the new array.)

The default array size is zero. This means that you need to call the `set...ArraySize()` before you can start filling array elements.

String members

You can declare string-valued fields with the following syntax:

```
message FooPacket
{
    string hostName;
};
```

The generated getter and setter methods will return and accept `const char*` pointers:

```
virtual const char *getHostName() const;
virtual void setHostName(const char *hostName);
```

The generated object will have its own copy of the string.

Note that a string member is different from a character array, which is treated as an array of any other type. For example,

```
message FooPacket
{
    char chars[10];
};
```

will generate the following methods:

```
virtual char getChars(unsigned k);
virtual void setChars(unsigned k, char a);
```

5.2.4 Inheritance, composition

So far we have discussed how to add fields of primitive types (**`int`**, **`double`**, **`char`**, ...) to `cMessage`. This might be sufficient for simple models, but if you have more complex models, you'll probably need to:

- set up a hierarchy of message (packet) classes, that is, not only subclass from `cMessage` but also from your own message classes;
- use not only primitive types as fields, but also structs, classes or typedefs. Sometimes you'll want to use a C++ type present in an already existing header file, another time you'll want a struct or class to be generated by the message compiler so that you can benefit from Tkenv inspectors.

The following section describes how to do this.

Inheritance among message classes

By default, messages are subclassed from `cMessage`. However, you can explicitly specify the base class using the

extends keyword:

```
message FooPacket extends FooBase
{
    ...
};
```

For the example above, the generated C++ code will look like:

```
class FooPacket : public FooBase { ... };
```

Inheritance also works for structs and classes (see next sections for details).

Defining classes

Until now we have used the **message** keyword to define classes, which implies that the base class is `cMessage`, either directly or indirectly.

But as part of complex messages, you'll need structs and other classes (rooted or not rooted in `cOwnedObject`) as building blocks. Classes can be created with the **class** class keyword; structs we'll cover in the next section.

The syntax for defining classes is almost the same as defining messages, only the **class** keyword is used instead of **message**.

Slightly different code is generated for classes that are rooted in `cOwnedObject` than for those which are not. If there is no `extends`, the generated class will not be derived from `cOwnedObject`, thus it will not have `getName()`, `getClassName()`, etc. methods. To create a class with those methods, you have to explicitly write `extends cOwnedObject`.

```
class MyClass extends cOwnedObject
{
    ...
};
```

Defining plain C structs

You can define C-style structs to be used as fields in message classes, ``C-style'' meaning ``containing only data and no methods''. (Actually, in the C++ a struct can have methods, and in general it can do anything a class can.)

The syntax is similar to that of defining messages:

```
struct MyStruct
{
    char array[10];
    short version;
};
```

However, the generated code is different. The generated struct has no getter or setter methods, instead the fields are represented by public data members. For the definition above, the following code is generated:

```
// generated C++
struct MyStruct
{
    char array[10];
    short version;
};
```

A struct can have primitive types or other structs as fields. It cannot have string or class as field.

Inheritance is supported for structs:

```
struct Base
{
    ...
};

struct MyStruct extends Base
{
    ...
};
```

But because a struct has no member functions, there are limitations:

- dynamic arrays are not supported (no place for the array allocation code)
- ``generation gap'' or abstract fields (see later) cannot be used, because they would build upon virtual functions.

Using structs and classes as fields

In addition to primitive types, you can also use other structs or objects as a field. For example, if you have a struct named `IPAddress`, you can write the following:

```
message FooPacket
{
    IPAddress src;
};
```

The `IPAddress` structure must be known in advance to the message compiler; that is, it must either be a struct or class defined earlier in the message description file, or it must be a C++ type with its header file included via `cplusplus {{...}}` and its type announced (see Announcing C++ types).

The generated class will contain an `IPAddress` data member (that is, **not** a pointer to an `IPAddress`). The following getter and setter methods will be generated:

```
virtual const IPAddress& getSrc() const;
virtual void setSrc(const IPAddress& src);
```

Pointers

Not supported yet.

5.2.5 Using existing C++ types

Announcing C++ types

If you want to use one of your own types (a class, struct or typedef, declared in a C++ header) in a message definition, you have to announce those types to the message compiler. You also have to make sure that your header file gets included into the generated `_m.h` file so that the C++ compiler can compile it.

Suppose you have an `IPAddress` structure, defined in an `ipaddress.h` file:

```
// ipaddress.h
struct IPAddress {
    int byte0, byte1, byte2, byte3;
};
```

To be able to use `IPAddress` in a message definition, the message file (say `foopacket.msg`) should contain the

following lines:

```
cplusplus {{
#include "ipaddress.h"
}};

struct IPAddress;
```

The effect of the first three lines is simply that the `#include` statement will be copied into the generated `foopacket_m.h` file to let the C++ compiler know about the `IPAddress` class. The message compiler itself will not try to make sense of the text in the body of the `cplusplus {{ ... }}` directive.

The next line, `struct IPAddress`, tells the message compiler that `IPAddress` is a C++ struct. This information will (among others) affect the generated code.

Classes can be announced using the **`class`** keyword:

```
class cSubQueue;
```

The above syntax assumes that the class is derived from `cOwnedObject` either directly or indirectly. If it is not, the **`noncobject`** keyword should be used:

```
class noncobject IPAddress;
```

The distinction between classes derived and not derived from `cOwnedObject` is important because the generated code differs at places. The generated code is set up so that if you incidentally forget the **`noncobject`** keyword (and thereby mislead the message compiler into thinking that your class is rooted in `cOwnedObject` when in fact it is not), you'll get a C++ compiler error in the generated header file.

5.2.6 Customizing the generated class

The Generation Gap pattern

Sometimes you need the generated code to do something more or do something differently than the version generated by the message compiler. For example, when setting a integer field named `payloadLength`, you might also need to adjust the packet length. That is, the following default (generated) version of the `setPayloadLength()` method is not suitable:

```
void FooPacket::setPayloadLength(int payloadLength)
{
    this->payloadLength = payloadLength;
}
```

Instead, it should look something like this:

```
void FooPacket::setPayloadLength(int payloadLength)
{
    int diff = payloadLength - this->payloadLength;
    this->payloadLength = payloadLength;
    setBitLength(length() + diff);
}
```

According to common belief, the largest drawback of generated code is that it is difficult or impossible to fulfill such wishes. Hand-editing of the generated files is worthless, because they will be overwritten and changes will be lost in the code generation cycle.

However, object oriented programming offers a solution. A generated class can simply be customized by subclassing from it and redefining whichever methods need to be different from their generated versions. This practice is known as the *Generation Gap* design pattern. It is enabled with the `@customize` property set on the message:

```
message FooPacket
{
   @customize(true);
   int payloadLength;
};
```

If you process the above code with the message compiler, the generated code will contain a `FooPacket_Base` class instead of `FooPacket`. Then you would subclass `FooPacket_Base` to produce `FooPacket`, while doing your customizations by redefining the necessary methods.

```
class FooPacket_Base : public cMessage
{
  protected:
    int src;
    // make constructors protected to avoid instantiation
    FooPacket_Base(const char *name=NULL);
    FooPacket_Base(const FooPacket_Base& other);
  public:
    ...
    virtual int getSrc() const;
    virtual void setSrc(int src);
};
```

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 is the following (you'll find it the generated C++ header too, as a comment):

```
class FooPacket : public FooPacket_Base
{
  public:
    FooPacket(const char *name=NULL) : FooPacket_Base(name) {}
    FooPacket(const FooPacket& other) : FooPacket_Base(other) {}
    FooPacket& operator=(const FooPacket& other)
        {FooPacket_Base::operator=(other); return *this;}
    virtual FooPacket *dup() {return new FooPacket(*this);}
};

Register_Class(FooPacket);
```

Note that it is important that you redefine `dup()` and provide an assignment operator (`operator=()`).

So, returning to our original example about payload length affecting packet length, the code you'd write is the following:

```
class FooPacket : public FooPacket_Base
{
    // here come the mandatory methods: constructor,
    // copy constructor, operator=(), dup()
    // ...

    virtual void setPayloadLength(int newlength);
}

void FooPacket::setPayloadLength(int newlength)
{
    // adjust message length
    setBitLength(length()-getPayloadLength()+newlength);
```

```
    // set the new length
    FooPacket_Base::setPayloadLength(newlength);
}
```

Abstract fields

The purpose of abstract fields is to let you to override the way the value is stored inside the class, and still benefit from inspectability in Tkenv.

For example, this is the situation when you want to store a bitfield in a single **`int`** or **`short`**, and still you want to present bits as individual packet fields. It is also useful for implementing computed fields.

You can declare any field to be abstract with the following syntax:

```
message FooPacket
{
   @customize(true);
   abstract bool urgentBit;
};
```

For an **`abstract`** field, the message compiler generates no data member, and generated getter/setter methods will be pure virtual:

```
virtual bool getUrgentBit() const = 0;
virtual void setUrgentBit(bool urgentBit) = 0;
```

Usually you'll want to use abstract fields together with the Generation Gap pattern, so that you can immediately redefine the abstract (pure virtual) methods and supply your implementation.

5.2.7 Using STL in message classes

You may want to use STL `vector` or `stack` classes in your message classes. This is possible using abstract fields. After all, `vector` and `stack` are representations of a *sequence* -- same abstraction as dynamic-size vectors. That is, you can declare the field as `abstract T fld[]`, and provide an underlying implementation using `vector<T>`. You can also add methods to the message class that invoke `push_back()`, `push()`, `pop()`, etc. on the underlying STL object.

See the following message declaration:

```
struct Item
{
    int a;
    double b;
}

message STLMessage
{
   @customize(true);
   abstract Item foo[]; // will use vector<Item>
   abstract Item bar[]; // will use stack<Item>
}
```

If you compile the above, in the generated code you'll only find a couple of abstract methods for `foo` and `bar`, no data members or anything concrete. You can implement everything as you like. You can write the following C++ file then to implement `foo` and `bar` with `std::vector` and `std::stack`:

```
#include <vector>
#include <stack>
```

```
#include "stlmessage_m.h"


class STLMessage : public STLMessage_Base
{
  protected:
    std::vector<Item> foo;
    std::stack<Item> bar;

  public:
    STLMessage(const char *name=NULL, int kind=0) : STLMessage_Base(name,kind) {}
    STLMessage(const STLMessage& other) : STLMessage_Base(other.getName())
{operator=(other);}
    STLMessage& operator=(const STLMessage& other) {
        if (&other==this) return *this;
        STLMessage_Base::operator=(other);
        foo = other.foo;
        bar = other.bar;
        return *this;
    }
    virtual STLMessage *dup() {return new STLMessage(*this);}

    // foo methods
    virtual void setFooArraySize(unsigned int size) {}
    virtual unsigned int getFooArraySize() const {return foo.size();}
    virtual Item& getFoo(unsigned int k) {return foo[k];}
    virtual void setFoo(unsigned int k, const Item& afoo) {foo[k]=afoo;}
    virtual void addToFoo(const Item& afoo) {foo.push_back(afoo);}

    // bar methods
    virtual void setBarArraySize(unsigned int size) {}
    virtual unsigned int getBarArraySize() const {return bar.size();}
    virtual Item& getBar(unsigned int k) {throw cRuntimeException("sorry");}
    virtual void setBar(unsigned int k, const Item& bar) {throw
cRuntimeException("sorry");}
    virtual void barPush(const Item& abar) {bar.push(abar);}
    virtual void barPop() {bar.pop();}
    virtual Item& barTop() {return bar.top();}
};

Register_Class(STLMessage);
```

Some additional notes:

- `setFooArraySize()`, `setBarArraySize()` are redundant.
- `getBar(int k)` cannot be implemented in a straightforward way (`std::stack` does not support accessing elements by index). It could still be implemented in a less efficient way using STL iterators, and efficiency does not seem to be major problem because only Tkenv is going to invoke this function.
- `setBar(int k, const Item&)` could not be implemented, but this is not particularly a problem. The exception will materialize in a Tkenv error dialog when you try to change the field value.

You may regret that the STL `vector/stack` are not directly exposed. Well you could expose them (by adding a `vector<Item>& getFoo() {return foo;}` method to the class) but this is probably not a good idea. STL itself was purposefully designed with a low-level approach, to provide ``nuts and bolts'' for C++ programming, and STL is better used in other classes for internal representation of data.

5.2.8 Summary

This section attempts to summarize the possibilities.

You can generate:

- classes rooted in `cOwnedObject`

- messages (default base class is `cMessage`)
- classes not rooted in `cOwnedObject`
- plain C structs

The following data types are supported for fields:

- primitive types: `bool`, `char`, `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `double`
- `string`, a dynamically allocated string, presented as `const char *`
- fixed-size arrays of the above types
- structs, classes (both rooted and not rooted in `cOwnedObject`), declared with the message syntax or externally in C++ code
- variable-sized arrays of the above types (stored as a dynamically allocated array plus an integer for the array size)

Further features:

- fields initialize to zero (except struct members)
- fields initializers can be specified (except struct members)
- assigning `enums` to variables of integral types.
- inheritance
- customizing the generated class via subclassing (*Generation Gap* pattern)
- abstract fields (for nonstandard storage and calculated fields)

Generated code (all generated methods are `virtual`, although this is not written out in the following table):

Field declaration	**Generated code**
primitive types `double field;`	`double getField();` `void setField(double d);`
string type `string field;`	`const char *getField();` `void setField(const char *);`

fixed-size arrays
`double field[4];`
	`double getField(unsigned k);`
	`void setField(unsigned k, double d);`
	`unsigned getFieldArraySize();`

dynamic arrays
`double field[];`
	`void setFieldArraySize(unsigned n);`
	`unsigned getFieldArraySize();`
	`double getField(unsigned k);`
	`void setField(unsigned k, double d);`

customized class
`class Foo {`
`  @customize(true);`
	`class Foo_Base { ... };`
	and you have to write:
	`class Foo : public Foo_Base {`
	`  ...`
	`};`

abstract fields `abstract double field;`	`double getField() = 0;` `void setField(double d) = 0;`

Example simulations

Several of the example simulations contain message definitions, for example Tictoc, Dyna, Routing and Hypercube. For example, in Dyna you'll find this:

- `dynapacket.msg` defines `DynaPacket` and `DynaDataPacket`;
- `dynapacket_m.h` and `dynapacket_m.cc` are produced by the message subclassing compiler from it, and they contain the generated `DynaPacket` and `DynaDataPacket` C++ classes (plus code for Tkenv inspectors);
- other model files (`client.cc`, `server.cc`, ...) use the generated message classes

5.2.9 What else is there in the generated code?

In addition to the message class and its implementation, the message compiler also generates reflection code which makes it possible to inspect message contents in Tkenv. To illustrate why this is necessary, suppose you manually subclass `cMessage` to get a new message class. You could write the following:

> [Note that the code is only for illustration. In real code, `freq` and `power` should be private members, and getter/setter methods should exist to access them. Also, the above class definition misses several member functions (constructor, assignment operator, etc.) that should be written.]

```
class RadioMsg : public cMessage
{
  public:
    int freq;
    double power;
    ...
};
```

Now it is possible to use `RadioMsg` in your simple modules:

```
RadioMsg *msg = new RadioMsg();
msg->freq = 1;
msg->power = 10.0;
...
```

You'll notice one drawback of this solution when you try to use Tkenv for debugging. While `cMsgPar`-based message parameters can be viewed in message inspector windows, fields added via subclassing do not appear there. The reason is that Tkenv, being just another C++ library in your simulation program, doesn't know about your C++ instance variables. The problem cannot be solved entirely within Tkenv, because C++ does not support ``reflection'' (extracting class information at runtime) like for example Java does.

There is a solution however: one can supply Tkenv with missing ``reflection'' information about the new class. Reflection info might take the form of a separate C++ class whose methods return information about the `RadioMsg` fields. This descriptor class might look like this:

```
class RadioMsgDescriptor : public Descriptor
{
  public:
    virtual int getFieldCount() {return 2;}

    virtual const char *getFieldName(int k) {
        const char *fieldname[] = {"freq", "power";}
        if (k<0 || k>=2) return NULL;
        return fieldname[k];
    }

    virtual double getFieldAsDouble(RadioMsg *msg, int k) {
        if (k==0) return msg->freq;
        if (k==1) return msg->power;
        return 0.0; // not found
```

```
    }
    //...
};
```

Then you have to inform Tkenv that a `RadioMsgDescriptor` exists and that it should be used whenever Tkenv finds messages of type `RadioMsg` (as it is currently implemented, whenever the object's `getClassName()` method returns `"RadioMsg"`). So when you inspect a `RadioMsg` in your simulation, Tkenv can use `RadioMsgDescriptor` to extract and display the values of the `freq` and `power` variables.

The actual implementation is somewhat more complicated than this, but not much.

---

6 The Simulation Library

OMNeT++ has an extensive C++ class library which you can use when implementing simple modules. Parts of the class library have already been covered in the previous chapters:

- the message class `cMessage` (chapter [5])
- sending and receiving messages, scheduling and canceling events, terminating the module or the simulation (section)
- access to module gates and parameters via `cModule` member functions (sections and)
- accessing other modules in the network (section)
- dynamic module creation (section)

This chapter discusses the rest of the simulation library:

- random number generation: `normal()`, `exponential()`, etc.
- module parameters: `cPar` class
- storing data in containers: the `cArray` and `cQueue` classes
- routing support and discovery of network topology: `cTopology` class
- recording statistics into files: `cOutVector` class
- collecting simple statistics: `cStdDev` and `cWeightedStddev` classes
- distribution estimation: `cLongHistogram`, `cDoubleHistogram`, `cVarHistogram`, `cPSquare`, `cKSplit` classes
- making variables inspectable in the graphical user interface (Tkenv): the `WATCH()` macros
- sending debug output to and prompting for user input in the graphical user interface (Tkenv): the `ev` object (`cEnvir` class)

6.1 Class library conventions

6.1.1 Base class

Classes in the OMNeT++ simulation library are derived from `cOwnedObject`. Functionality and conventions that come from `cOwnedObject`:

- name attribute
- `getClassName()` member and other member functions giving textual information about the object
- conventions for assignment, copying, duplicating the object
- ownership control for containers derived from `cOwnedObject`
- support for traversing the object tree
- support for inspecting the object in graphical user interfaces (Tkenv)

Classes inherit and redefine several `cOwnedObject` member functions; in the following we'll discuss some of the practically important ones.

6.1.2 Setting and getting attributes

Member functions that set and query object attributes follow a naming convention: the setter member function begins with `set`, and the getter begins with `get` (or in the case of boolean attributes, with `is` or `has`, whichever is more appropriate). For example, the *length* attribute of the `cPacket` class can be set and read like this:

```
pk->setBitLength(1024);
length = pk->getBitLength();
```

> **NOTE**
> OMNeT++ 3.x and earlier versions did not have the `get` verb in the name of getter methods. There are scripts to port old source code to OMNeT++ 4.0; these tools and the suggested porting produre are described in the *Migration Guide*.

6.1.3 getClassName()

For each class, the `getClassName()` member function returns the class name as a string:

```
const char *classname = msg->getClassName(); // returns "cMessage"
```

6.1.4 Name attribute

An object can be assigned a *name* (a character string). The name string is the first argument to the constructor of every class, and it defaults to `NULL` (no name string). An example:

```
cMessage *timeoutMsg = new cMessage("timeout");
```

You can also set the name after the object has been created:

```
timeoutMsg->setName("timeout");
```

You can get a pointer to the internally stored copy of the name string like this:

```
const char *name = timeoutMsg->getName(); // --> "timeout"
```

For convenience and efficiency reasons, the empty string `""` and `NULL` are treated as equivalent by library objects. That is, `""` is stored as `NULL` but returned as `""`. If you create a message object with either `NULL` or `""` as name string, it will be stored as `NULL` and `getName()` will return a pointer to a static `""`.

```
cMessage *msg = new cMessage(NULL, <additional args>);
const char *str = msg->getName(); // --> returns ""
```

6.1.5 getFullName() and getFullPath()

Objects have two more member functions which return strings based on object names: `getFullName()` and `getFullPath()`. For gates and modules which are part of gate or module vectors, `getFullName()` returns the name with the index in brackets. That is, for a module `node[3]` in the submodule vector `node[10]` `getName()`

returns "node", and getFullName() returns "node[3]". For other objects, getFullName() is the same as getName().

getFullPath() returns getFullName(), prepended with the parent or owner object's getFullPath() and separated by a dot. That is, if the node[3] module above is in the compound module "net.subnet1", its getFullPath() method will return "net.subnet1.node[3]".

```
ev << this->getName();     // --> "node"
ev << this->getFullName(); // --> "node[3]"
ev << this->getFullPath(); // --> "net.subnet1.node[3]"
```

getClassName(), getFullName() and getFullPath() are extensively used on the graphical runtime environment Tkenv, and also appear in error messages.

getName() and getFullName() return const char * pointers, and getFullPath() returns std::string. This makes no difference with ev<<, but when getFullPath() is used as a "%s" argument to sprintf() you have to write getFullPath().c_str().

```
char buf[100];
sprintf("msg is '%80s'", msg->getFullPath().c_str()); // note c_str()
```

6.1.6 Copying and duplicating 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, which in turn relies on the assignment operator between objects. operator=() can be used to copy contents of an object into another object of the same type. This is a deep copy: object contained in the object will also be duplicated if necessary. operator=() does not copy the name string -- this task is done by the copy constructor.

> **NOTE**
> Since the OMNeT++ 4.0 version, dup() returns a pointer to the same type as the object itself, and not a cObject*. This is made possible by a relatively new C++ feature called *covariant return types*.

6.1.7 Iterators

There are several container classes in the library (cQueue, cArray etc.) For many of them, there is a corresponding iterator class that you can use to loop through the objects stored in the container.

For example:

```
cQueue queue;

//..
for (cQueue::Iterator queueIter(queue); !queueIter.end(); queueIter++)
{
    cOwnedObject *containedObject = queueIter();
}
```

6.1.8 Error handling

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` ini file entry lets you do that -- check it in section .

If you detect an error condition in your code, you can stop the simulation with an error message using the `opp_error()` function. `opp_error()`'s argument list works like `printf()`: the first argument is a format string which can contain `"%s"`, `"%d"` etc, filled in using subsequent arguments.

An example:

```
if (msg->getControlInfo()==NULL)
    opp_error("message (%s)%s has no control info attached",
              msg->getClassName(), msg->getName());
```

6.2 Logging from modules

The logging feature will be used extensively in the code examples, we introduce it here.

The `ev` object represents the user interface of the simulation. You can send debugging output to `ev` with the C++-style output operators:

```
ev << "packet received, sequence number is " << seqNum << endl;
ev << "queue full, discarding packet\n";
```

An alternative solution is `ev.printf()`:

```
ev.printf("packet received, sequence number is %d\n", seqNum);
```

The exact way messages are displayed to the user depends on the user interface. In the command-line user interface (Cmdenv), it is simply dumped to the standard output. (This output can also be disabled from `omnetpp.ini` so that it doesn't slow down simulation when it is not needed.) In Tkenv, the runtime GUI, you can open a text output window for every module. It is not recommended that you use `printf()` or `cout` to print messages -- `ev` output can be controlled much better from `omnetpp.ini` and it is more convenient to view, using Tkenv.

One can save CPU cycles by making logging statements conditional on whether the output actually gets displayed or recorded anywhere. The `ev.isDisabled()` call returns true when `ev<<` output is disabled, such as in Tkenv or Cmdenv ``express'' mode. Thus, one can write code like this:

```
if (!ev.isDisabled())
    ev << "Packet " << msg->getName() << " received\n";
```

A more sophisticated implementation of the same idea is to the `EV` macro which can be used in logging statements instead of `ev`. One would simply write `EV<<` instead of `ev<<`:

```
EV << "Packet " << msg->getName() << " received\n";
```

`EV`'s implementation makes use of the fact that the `<<` operator binds looser than the conditional operator (`?:`).

6.3 Simulation time conversion

Simulation time is represented by the type `simtime_t` which is a typedef to `double`. OMNeT++ provides utility

functions, which convert `simtime_t` to a printable string (`"3s 130ms 230us"`) and vica versa.

The `simtimeToStr()` function converts a `simtime_t` (passed in the first argument) to textual form. The result is placed into the `char` array pointed to by the second argument. If the second argument is omitted or it is `NULL`, `simtimeToStr()` will place the result into a static buffer which is overwritten with each call. An example:

```
char buf[32];
ev.printf("t1=%s, t2=%s\n", simtimeToStr(t1), simTimeToStr(t2,buf));
```

The `simtimeToStrShort()` is similar to `simtimeToStr()`, but its output is more concise.

The `strToSimtime()` function parses a time specification passed in a string, and returns a `simtime_t`. If the string cannot be entirely interpreted, -1 is returned.

```
simtime_t t = strToSimtime("30s 152ms");
```

Another variant, `strToSimtime0()` can be used if the time string is a substring in a larger string. Instead of taking a `char*`, it takes a reference to `char*` (`char*&`) as the first argument. The function sets the pointer to the first character that could not be interpreted as part of the time string, and returns the value. It never returns -1; if nothing at the beginning of the string looked like simulation time, it returns 0.

```
const char *s = "30s 152ms and something extra";

simtime_t t = strToSimtime0(s); // now s points to "and something extra"
```

6.4 Generating random numbers

Random numbers in simulation are never random. Rather, they are produced using deterministic algorithms. Algorithms take a *seed* value and perform some deterministic calculations on them to produce a ``random'' number and the next seed. 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.

> [There are real random numbers as well, see e.g. http://www.random.org/, http://www.comscire.com, or the Linux */dev/random* device. For non-random numbers, try www.noentropy.net.]

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 produce uniformly distributed integers in some range, usually between 0 or 1 and 2^{32} or so. Mathematical transformations are used to produce random variates from them that correspond to specific distributions.

6.4.1 Random number generators

Mersenne Twister

By default, OMNeT++ uses the Mersenne Twister RNG (MT) by M. Matsumoto and T. Nishimura [Matsumoto98]. 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

OMNeT++ releases prior to 3.0 used a linear congruential generator (LCG) with a cycle length of $2^{31}-2$, described in [Jain91], pp. 441-444,455. This RNG is still available and can be selected from `omnetpp.ini` (Chapter [9]).

This RNG is only suitable for small-scale simulation studies. As shown by Karl Entacher et al. in [Entacher02], 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 [Hellekalek98] 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 you execute simulations under Akaroa control (see section [9.5]), you can also select Akaroa's RNG as the RNG underlying for the OMNeT++ random number functions. The Akaroa RNG also has to be selected from `omnetpp.ini` (section [8.7]).

Other RNGs

OMNeT++ allows plugging in your own RNGs as well. This mechanism, based on the `cRNG` interface, is described in section . For example, one candidate to include could be L'Ecuyer's CMRG [LEcuyer02] which has a period of about 2^{191} and can provide a large number of *guaranteed* independent streams.

6.4.2 Random number streams, RNG mapping

Simulation programs may consume random numbers from several streams, that is, from several independent RNG instances. For example, if a network simulation uses random numbers for generating packets and for simulating bit errors in the transmission, it might be a good idea to use different random streams for both. Since the seeds for each stream can be configured independently, this arrangement would allow you 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.

It is also important that different streams and also different simulation runs use non-overlapping series of random numbers. Overlap in the generated random number sequences can introduce unwanted correlation in your results.

The number of random number streams as well as seeds for the individual streams can be configured in `omnetpp.ini` (section [8.7]). For the "minimal standard RNG", the `seedtool` program can be used for selecting good seeds (section).

In OMNeT++, streams are identified with RNG numbers. The RNG numbers used in simple modules may be *arbitrarily mapped* to the actual random number streams (actual RNG instances) from `omnetpp.ini` (section [8.7]). 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.

6.4.3 Accessing the RNGs

The `intrand(n)` function generates random integers in the range *[0, n-1]*, and `dblrand()` generates a random double on *[0,1)*. These functions simply wrap the underlying RNG objects. Examples:

```
int dice = 1 + intrand(6); // result of intrand(6) is in the range 0..5
double p = dblrand();      // dblrand() produces numbers in [0,1)
```

They also have a counterparts that use generator *k*:

```
int dice = 1 + genk_intrand(k,6); // uses generator k
double prob = genk_dblrand(k);    // ""
```

The underlying RNG objects are subclassed from `cRNG`, and they can be accessed via `cModule`'s `getRNG()` method. The argument to `getRNG()` is a local RNG number which will undergo RNG mapping.

```
cRNG *rng1 = getRNG(1);
```

cRNG contains the methods implementing the above intrand() and dblrand() functions. The cRNG interface also allows you to access the ``raw'' 32-bit random numbers generated by the RNG and to learn their ranges (intRand(), intRandMax()) as well as to query the number of random numbers generated (getNumbersDrawn()).

6.4.4 Random variates

The following functions are based on dblrand() and return random variables of different distributions:

Random variate functions use one of the random number generators (RNGs) provided by OMNeT++. By default this is generator 0, but you can specify which one to be used.

OMNeT++ has the following predefined distributions:

Function	Description
Continuous distributions	
uniform(a, b, *rng=0*)	uniform distribution in the range [a,b)
exponential(mean, *rng=0*)	exponential distribution with the given mean
normal(mean, stddev, *rng=0*)	normal distribution with the given mean and standard deviation
truncnormal(mean, stddev, *rng=0*)	normal distribution truncated to nonnegative values
gamma_d(alpha, beta, *rng=0*)	gamma distribution with parameters alpha>0, beta>0
beta(alpha1, alpha2, *rng=0*)	beta distribution with parameters alpha1>0, alpha2>0
erlang_k(k, mean, *rng=0*)	Erlang distribution with k>0 phases and the given mean
chi_square(k, *rng=0*)	chi-square distribution with k>0 degrees of freedom
student_t(i, *rng=0*)	student-t distribution with i>0 degrees of freedom
cauchy(a, b, *rng=0*)	Cauchy distribution with parameters a,b where b>0
triang(a, b, c, *rng=0*)	triangular distribution with parameters a<=b<=c, a!=c
lognormal(m, s, rng=0)	lognormal distribution with mean m and variance s>0
weibull(a, b, *rng=0*)	Weibull distribution with parameters a>0, b>0
pareto_shifted(a, b, c, *rng=0*)	generalized Pareto distribution with parameters a, b and shift c
Discrete distributions	
intuniform(a, b, *rng=0*)	uniform integer from a..b
bernoulli(p, *rng=0*)	result of a Bernoulli trial with probability 0<=p<=1 (1 with probability p and 0 with probability (1-p))
binomial(n, p, *rng=0*)	binomial distribution with parameters n>=0 and 0<=p<=1
geometric(p, *rng=0*)	geometric distribution with parameter 0<=p<=1
negbinomial(n, p, *rng=0*)	binomial distribution with parameters n>0 and 0<=p<=1
poisson(lambda, *rng=0*)	Poisson distribution with parameter lambda

They are the same functions that can be used in NED files. 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.

If the above distributions do not suffice, you can write your own functions. If you register your functions with the `Register_Function()` macro, you can use them in NED files and ini files too.

6.4.5 Random numbers from histograms

You can also specify your distribution as a histogram. The `cLongHistogram`, `cDoubleHistogram`, `cVarHistogram`, `cKSplit` or `cPSquare` classes are there to generate random numbers from equidistant-cell or equiprobable-cell histograms. This feature is documented later, with the statistical classes.

6.5 Container classes

6.5.1 Queue class: cQueue

Basic usage

`cQueue` is a container class that acts as a queue. `cQueue` can hold objects of type derived from `cOwnedObject` (almost all classes from the OMNeT++ library), such as `cMessage`, `cPar`, etc. Internally, `cQueue` uses a double-linked list to store the elements.

A queue object has a head and a tail. Normally, new elements are inserted at its head and elements are removed at its tail.

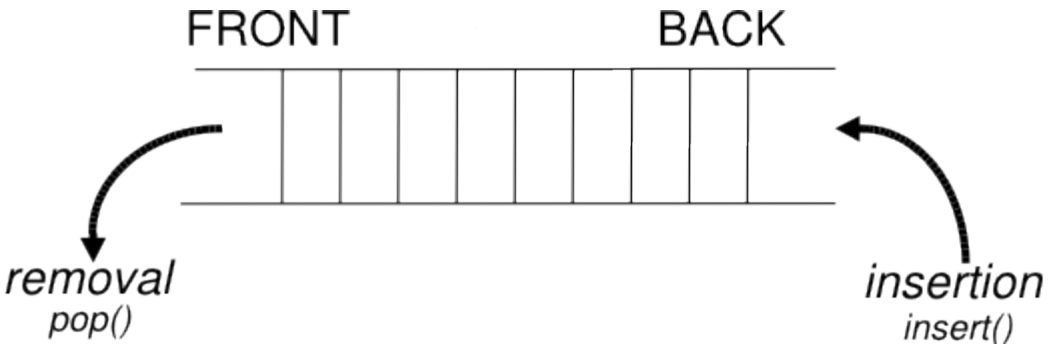


Figure: cQueue: insertion and removal

The basic `cQueue` member functions dealing with insertion and removal are `insert()` and `pop()`. They are used like this:

```
cQueue queue("my-queue");
cMessage *msg;

// insert messages
for (int i=0; i<10; i++)
{
  msg = new cMessage;
  queue.insert(msg);
}

// remove messages
while(!queue.empty())
{
  msg = (cMessage *)queue.pop();
  delete msg;
}
```

The `length()` member function returns the number of items in the queue, and `empty()` tells whether there's

anything in the queue.

There are other functions dealing with insertion and removal. The `insertBefore()` and `insertAfter()` functions insert a new item exactly before and 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. If you want to use this feature, you have to provide a function that takes two `cOwnedObject` pointers, compares the two objects and returns -1, 0 or 1 as the result (see the reference for details). An example of setting up an ordered `cQueue`:

```
cQueue sortedqueue("sortedqueue", cOwnedObject::cmpbyname, true );
                        // sorted by object name, ascending
```

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

Normally, you can only access the objects at the head or tail of the queue. However, if you use an iterator class, `cQueue::Iterator`, you can examine each object in the queue.

The `cQueue::Iterator` constructor takes two arguments, the first is the queue object and the second one specifies the initial position of the iterator: 0=tail, 1=head. Otherwise it acts as any other OMNeT++ iterator class: you 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 if you're at the end (or the beginning) of the queue.

An example:

```
for( cQueue::Iterator iter(queue,1); !iter.end(), iter++)
{
  cMessage *msg = (cMessage *) iter();
  //...
}
```

6.5.2 Expandable array: cArray

Basic usage

`cArray` is a container class that holds objects derived from `cOwnedObject`. `cArray` stores the pointers of the objects inserted instead of making copies. `cArray` works as an array, but it grows automatically when it gets full. Internally, `cArray` is implemented with an array of pointers; when the array fills up, it is reallocated.

`cArray` objects are used in OMNeT++ to store parameters attached to messages, and internally, for storing module parameters and gates.

Creating an array:

```
cArray array("array");
```

Adding an object at the first free index:

```
cPar *p = new cMsgPar("par");
int index = array.add( p );
```

Adding an object at a given index (if the index is occupied, you'll get an error message):

```
cPar *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?
  {
    cOwnedObject *obj = array[i];
    ev << obj->getName() << endl;
  }
}
```

6.6 Routing support: cTopology

6.6.1 Overview

The `cTopology` class was designed primarily to support routing in telecommunication or multiprocessor networks.

A `cTopology` object stores an abstract representation of the network in 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*.

You can specify which modules (either simple or compound) you want to include in the graph. The graph will include all connections among the selected modules. In the graph, all nodes are at the same level, there's 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're 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: you can easily find the corresponding module for a `cTopology` node and vica versa.

6.6.2 Basic usage

You can extract the network topology into a `cTopology` object by a single function call. You have several ways to select which modules you want to include in the topology:

- by module type
- by a parameter's presence and its 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", NULL);
```

Any number of module types can be supplied; the list must be terminated by `NULL`.

A dynamically assembled list of module types can be passed as a `NULL`-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] = NULL;
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() == simulation.getSystemModule();
}

topo.extractFromNetwork( selectFunction, NULL );
```

A cTopology object uses two types: cTopology::Node for nodes and cTopology::Link for edges. (sTopoLinkIn 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 (1st line) returns the number of nodes in the graph, and getNode(i) returns a pointer to the *i*th node, an cTopology::Node structure.

The correspondence between a graph node and a module can be obtained by:

```
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 NULL). getNodeFor() uses binary search within the cTopology object so it is fast enough.

cTopology::Node's other member functions let you determine the connections of this node: getNumInLinks(), getNumOutLinks() return the number of connections, in(i) and out(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.)

6.6.3 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 when we have the target module pointer, finding the shortest path 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 == NULL)
{
  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 calculate this, you calculate the shortest paths to the target *from the neighbor node*, but you must disable 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);
```

6.7 Statistics and distribution estimation

6.7.1 cStatistic and descendants

There are several statistic and result collection classes: `cStdDev`, `cWeightedStdDev`, `LongHistogram`, `cDoubleHistogram`, `cVarHistogram`, `cPSquare` and `cKSplit`. They are all derived from the abstract base class `cStatistic`.

- `cStdDev` keeps the count, mean, standard deviation, minimum and maximum value etc of the observations.
- `cWeightedStdDev` is similar to `cStdDev`, but accepts weighted observations. `cWeightedStdDev` can be used for example to calculate time average. It is the only weighted statistics class.
- `cLongHistogram` and `cDoubleHistogram` are descendants of `cStdDev` and also keep an approximation of the distribution of the observations using equidistant (equal-sized) cell histograms.
- `cVarHistogram` implements a histogram where cells do not need to be the same size. You can manually add the cell (bin) boundaries, or alternatively, automatically have a partitioning created where each bin has the same number of observations (or as close to that as possible).
- `cPSquare` is a class that uses the P^2 algorithm described in [JCh85]. The algorithm calculates quantiles without storing the observations; one can also think of it as a histogram with equiprobable cells.
- `cKSplit` uses a novel, experimental method, based on an adaptive histogram-like algorithm.

Basic usage

One can insert an observation into a statistic object with the `collect()` function or the += operator (they are equivalent). `cStdDev` has the following methods for getting statistics out of the object: `getCount()`, `getMin()`, `getMax()`, `getMean()`, `getStddev()`, `getVariance()`, `getSum()`, `getSqrSum()` with the obvious meanings. An example usage for `cStdDev`:

```
cStdDev stat("stat");

for (int i=0; i<10; i++)
  stat.collect( normal(0,1) );

long numSamples = stat.getCount();
double smallest = stat.getMin(),
       largest = stat.getMax();
double mean = stat.getMean(),
       standardDeviation = stat.getStddev(),
       variance = stat.getVariance();
```

6.7.2 Distribution estimation

Initialization and usage

The distribution estimation classes (`cLongHistogram`, `cDoubleHistogram`, `cVarHistogram`, `cPSquare` and `cKSplit`) are derived from `cDensityEstBase`. Distribution estimation classes (except for `cPSquare`) assume that the observations are within a range. You may specify the range explicitly (based on some a-priori info about the distribution) or you may let the object collect the first few observations and determine the range from them. Methods which let you specify range settings are part of `cDensityEstBase`.

The following member functions exist for setting up the range and to specify how many observations should be used for automatically determining the range.

```
setRange(lower,upper);
setRangeAuto(numFirstvals, rangeExtFactor);
setRangeAutoLower(upper, numFirstvals, rangeExtFactor);
setRangeAutoUpper(lower, numFirstvals, rangeExtFactor);
```

```
setNumFirstVals(numFirstvals);
```

The following example creates a histogram with 20 cells and automatic range estimation:

```
cDoubleHistogram histogram("histogram", 20);
histogram.setRangeAuto(100,1.5);
```

Here, 20 is the number of cells (not including the underflow/overflow cells, see later), and 100 is the number of observations to be collected before setting up the cells. 1.5 is the range extension factor. It means that the actual range of the initial observations will be expanded 1.5 times and this expanded range will be used to lay out the cells. This method increases the chance that further observations fall in one of the cells and not outside the histogram range.

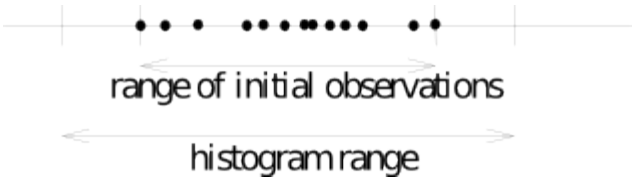


Figure: Setting up a histogram's range

After the cells have been set up, collection can go on.

The `isTransformed()` function returns *true* when the cells have already been set up. You can force range estimation and setting up the cells by calling the `transform()` function.

The observations that fall outside the histogram range will be counted as underflows and overflows. The number of underflows and overflows are returned by the `getUnderflowCell()` and `getOverflowCell()` member functions.

Figure: Histogram structure after setting up the cells

You create a P^2 object by specifying the number of cells:

```
cPSquare psquare("interarrival-times", 20);
```

Afterwards, a `cPSquare` can be used with the same member functions as a histogram.

Getting histogram data

There are three member functions to explicitly return cell boundaries and the number of observations is each cell.

`getNumCells()` returns the number of cells, `getBasepoint(int k)` returns the *k*th base point, `getCellValue(int k)` returns the number of observations in cell *k*, and `getCellPDF(int k)` returns the PDF value in the cell (i.e. between `getBasepoint(k)` and `getBasepoint(k+1)`). The `getCellInfo(k)` method returns multiple data (cell bounds, counter, relatile frequency) packed together in a struct. These functions work for all histogram types, plus `cPSquare` and `cKSplit`.

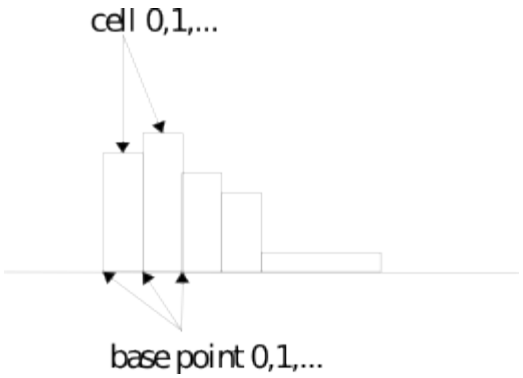


Figure: base points and cells

An example:

```
long n = histogram.getCount();
for (int i=0; i<histogram.getNumCells(); i++)
{
  double cellWidth = histogram.getBasepoint(i+1)-histogram.getBasepoint(i);
  int count = histogram.getCellValue(i);
  double pdf = histogram.getCellPDF(i);
  //...
}
```

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.

Random number generation from distributions

The `random()` member function generates random numbers from the distribution stored by the object:

```
double rnd = histogram.random();
```

`cStdDev` assumes normal distribution.

You can also wrap the distribution object in a `cPar`:

```
cMsgPar rndPar("rndPar");
rndPar.setDoubleValue(&histogram);
```

The `cPar` object stores the pointer to the histogram (or P^2 object), and whenever it is asked for the value, calls the histogram object's `random()` function:

```
double rnd = (double)rndPar; // random number from the cPSquare
```

Storing/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);

cDoubleHistogram hist2("Hist-from-file");
FILE *f2 = fopen("histogram.dat","r");
hist2.loadFromFile(f2); // load stored distribution
fclose(f2);
```

Histogram with custom cells

The `cVarHistogram` class can be used to create histograms with arbitrary (non-equidistant) cells. It can operate in two modes:

- *manual*, where you specify cell boundaries explicitly before starting collecting
- *automatic*, where `transform()` will set up the cells after collecting a certain number of initial observations. The cells will be set up so that as far as possible, an equal number of observations fall into each cell (equi-probable cells).

Modes are selected with a *transform-type* parameter:

- `HIST_TR_NO_TRANSFORM`: no transformation; uses bin boundaries previously defined by `addBinBound()`
- `HIST_TR_AUTO_EPC_DBL`: automatically creates equiprobable cells
- `HIST_TR_AUTO_EPC_INT`: like the above, but for integers

Creating an object:

```
cVarHistogram(const char *s=NULL,
              int numcells=11,
              int transformtype=HIST_TR_AUTO_EPC_DBL);
```

Manually adding a cell boundary:

```
void addBinBound(double x);
```

Rangemin and rangemax is chosen after collecting the `numFirstVals` initial observations. One cannot add cell boundaries when the histogram has already been transformed.

6.7.3 The k-split algorithm

Purpose

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 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 $n_1, n_2, .. n_k$. 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 $n_{i,1}, n_{i,2}, .. n_{i,k}$ initialized to zero. The n 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 shows that *k=2* worked best.

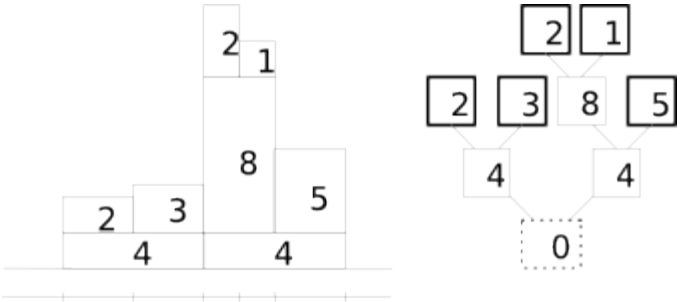


Figure: Illustration of the k-split algorithm, k=2. The numbers in boxes represent the observation count values

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 $?_{...,i} = n_{...,i} + m_{...,i}$ estimated amount of observations in the tree nodes, especially in the leaves. In other words, if we have $?_{...,i}$ estimated observation amount in a cell, how to divide it to obtain $m_{...,i,1}$, $m_{...,i,2}$ .. $m_{...,i,k}$ that can be propagated to child cells. Naturally, $m_{...,i,1} + m_{...,i,2} + .. + m_{...,i,k} = ?_{...,i}$.

Two natural distribution methods are even distribution (when $m_{...,i,1} = m_{...,i,2} = .. = m_{...,i,k}$) and proportional distribution (when $m_{...,i,1} : m_{...,i,2} : .. : m_{...,i,k} = s_{...,i,1} : s_{...,i,2} : .. : s_{...,i,k}$). 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 *λ=0* means even and *λ=1* means proportional distribution:

$m_{..,i,j} = (1-\lambda)?_{..,i}/k + \lambda\ ?_{..,i}\ s_{...,i,j} / s_{..,i}$ where *λ is in [0,1]*

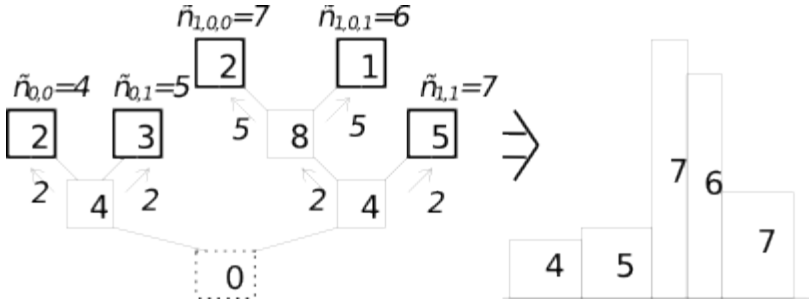


Figure: Density estimation from the k-split cell tree. We assume λ=0, i.e. we distribute mother observations evenly.

Note that while $n_{...,i}$ are integers, $m_{...,i}$ and thus $?_{...,i}$ 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.

OMNeT++ contains an experimental implementation of the *k*-split algorithm, the `cKSplit` class. Research on *k*-split is still under way.

The cKSplit class

The `cKSplit` class is an implementation of the *k-split* method. Member functions:

```
void setCritFunc(KSplitCritFunc _critfunc, double *_critdata);
void setDivFunc(KSplitDivFunc \_divfunc, double *\_divdata);
void rangeExtension( bool enabled );
```

```
int getTreeDepth();
int getTreeDepth(cKSplit::Grid& grid);
```

```
double getRealCellValue(cKSplit::Grid& grid, int cell);
void printGrids();
```

```
cKSplit::Grid& getGrid(int k);
cKSplit::Grid& getRootGrid();
```

```
struct cKSplit::Grid
{
  int parent;   // index of parent grid
  int reldepth; // depth = (reldepth - rootgrid's reldepth)
  long total;   // sum of cells & all subgrids (includes "mother")
  int mother;   // observations "inherited" from mother cell
  int cells[K]; // cell values
};
```

6.7.4 Transient detection and result accuracy

In many simulations, only the steady state performance (i.e. the performance after the system has reached a stable state) is of interest. The initial part of the simulation is called the transient period. After the model has entered steady state, simulation must proceed until enough statistical data has been collected to compute result with the required accuracy.

Detection of the end of the transient period and a certain result accuracy is supported by OMNeT++. The user can attach transient detection and result accuracy objects to a result object (`cStatistic`'s descendants). The transient detection and result accuracy objects will do the specific algorithms on the data fed into the result object and tell if the transient period is over or the result accuracy has been reached.

The base classes for classes implementing specific transient detection and result accuracy detection algorithms are:

- `cTransientDetection`: base class for transient detection
- `cAccuracyDetection`: base class for result accuracy detection

Basic usage

Attaching detection objects to a `cStatistic` and getting pointers to the attached objects:

```
addTransientDetection(cTransientDetection *object);
addAccuracyDetection(cAccuracyDetection *object);
cTransientDetection *getTransientDetectionObject();
cAccuracyDetection *getAccuracyDetectionObject();
```

Detecting the end of the period:

- polling the `detect()` function of the object
- installing a post-detect function

Transient detection

Currently one transient detection algorithm is implemented, i.e. there's one class derived from `cTransientDetection`. The `cTDExpandingWindows` class uses the sliding window approach with two windows, and checks the difference of the two averages to see if the transient period is over.

```
void setParameters(int reps=3,
                   int minw=4,
                   double wind=1.3,
                   double acc=0.3);
```

Accuracy detection

Currently one accuracy detection algorithm is implemented, i.e. there's one class derived from `cAccuracyDetection`. The algorithm implemented in the `cADByStddev` class is: divide the standard deviation by the square of the number of values and check if this is small enough.

```
void setParameters(double acc=0.1, int reps=3);
```

6.8 Recording simulation results

6.8.1 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* named `omnetpp.vec` by default. You can configure output vectors from `omnetpp.ini`: you can disable writing to the file, or limit it to a certain simulation time interval for recording (section [8.6.2]).

The format and processing of output vector files is described in section [11.1.2].

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 Tkenv inspector window open for the output vector object, the values will be displayed there, regardless of the state of the output vector object.

6.8.2 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'll 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 statistics 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()
{
    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 write into the *output scalar file* which is named `omnetpp.sca` by default. The output scalar file is preserved across simulation runs (unlike the output vector file which gets deleted at the beginning of every simulation run). Data are always appended at the end of the file, and output from different simulation runs are separated by special lines. The format and processing of output vector files is described in section .

6.8.3 Precision

Output scalar and output vector files are text files, and floating point values (`doubles`) 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 entries (see). The default precision is 12 digits. The following has to be considered when changing the default 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 an IEEE-754 double (as in Plove and Scalars), an additional small error will occurs because 0.1, 0.01, etc cannot be accurately represented in binary. This conversion error is usually smaller than the one that the `double` variable already had before recording into the file, however if it is important you can eliminate it by setting >16 digits precision for the file (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 coming from converting to/from decimal representation can be eliminated by choosing an output vector/output scalar manager class which stores `doubles` in their native binary form. The appropriate configuration entries are `outputvectormanager-class=` and `outputvectormanager-class=`; see . For example, `cMySQLOutputScalarManager` and `cMySQLOutputScalarManager` provided in `samples/database` fulfill this requirement.

However, before worrying too much about rounding and conversion errors, it is worth considering what is the *real* accuracy of your results. Some things to consider:

- in real life, it is very hard 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?

6.9 Watches and snapshots

6.9.1 Basic watches

It would be nice, but variables of type `int`, `long`, `double` do not show up by default in Tkenv; 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.

OMNeT++ provides `WATCH()` and set of other macros to come to your rescue, and make variable to be inspectable in Tkenv 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);
```

When you open an inspector for the simple module in Tkenv and click the Objects/Watches tab in it, you'll see your watched variables and their values there. Tkenv also lets you change the value of a watched variable.

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);
```

6.9.2 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);
```

6.9.3 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 one can do better. 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 `NULL` at all times, otherwise the GUI may crash while trying to display the object. This practically means that the pointer should be initialized to `NULL` even if not used, and should be set to `NULL` when the object to which it points gets deleted.

```
delete watchedPtr;
watchedPtr = NULL;  // set to NULL when object gets deleted
```

6.9.4 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_PTRSET()`, `WATCH_MAP()`, `WATCH_PTRMAP()`.

The `PTR`-less versions expect the data items ("T") to have stream output operators (`operator <<`), because that's 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);
```

6.9.5 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(cOwnedObject *obj = &simulation, const char *label = NULL);
```

The function can be called from module functions, like this:

```
snapshot();     // dump the whole network
snapshot(this); // dump this simple module and all its objects
snapshot(&simulation.msgQueue); // dump future events
```

This will append snapshot information to the end of the snapshot file. (The snapshot file name has an extension of `.sna`, default is `omnetpp.sna`. Actual file name can be set in the config file.)

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 Tkenv, you can also create a snapshot from the menu.

An example of a snapshot file:

```
[...]

(cSimulation) 'simulation' begin
  Modules in the network:
    'token' #1 (TokenRing)
      'comp[0]' #2 (Computer)
        'mac' #3 (TokenRingMAC)
        'gen' #4 (Generator)
        'sink' #5 (Sink)
      'comp[1]' #6 (Computer)
        'mac' #7 (TokenRingMAC)
        'gen' #8 (Generator)
        'sink' #9 (Sink)
      'comp[2]' #10 (Computer)
        'mac' #11 (TokenRingMAC)
        'gen' #12 (Generator)
        'sink' #13 (Sink)
end

(TokenRing) 'token' begin
  #1 params     (cArray) (n=6)
  #1 gates      (cArray) (empty)
  comp[0]          (cCompoundModule,#2)
  comp[1]          (cCompoundModule,#6)
  comp[2]          (cCompoundModule,#10)
end

(cArray) 'token.parameters' begin
  num_stations (cModulePar) 3 (L)
  num_messages (cModulePar) 10000 (L)
  ia_time      (cModulePar) truncnormal(0.005,0.003) (F)
  THT          (cModulePar) 0.01 (D)
  data_rate    (cModulePar) 4000000 (L)
  cable_delay  (cModulePar) 1e-06 (D)
end

[...]

(cQueue) 'token.comp[0].mac.local-objects.send-queue' begin
  0-->1         (cMessage) Tarr=0.0158105774 ( 15ms) Src=#4 Dest=#3
```

```
  0-->2         (cMessage) Tarr=0.0163553310 ( 16ms) Src=#4 Dest=#3
  0-->1         (cMessage) Tarr=0.0205628236 ( 20ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0242203591 ( 24ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0300994268 ( 30ms) Src=#4 Dest=#3
  0-->1         (cMessage) Tarr=0.0364005251 ( 36ms) Src=#4 Dest=#3
  0-->1         (cMessage) Tarr=0.0370745702 ( 37ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0387984129 ( 38ms) Src=#4 Dest=#3
  0-->1         (cMessage) Tarr=0.0457462493 ( 45ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0487308918 ( 48ms) Src=#4 Dest=#3
  0-->2         (cMessage) Tarr=0.0514466766 ( 51ms) Src=#4 Dest=#3
end

(cMessage) 'token.comp[0].mac.local-objects.send-queue.0-->1' begin
  #4 --> #3
  sent:          0.0158105774 ( 15ms)
  arrived:       0.0158105774 ( 15ms)
  length:        33536
  kind:          0
  priority:      0
  error:         FALSE
  time stamp:    0.0000000 ( 0.00s)
  parameter list:
    dest         (cPar) 1 (L)
    source       (cPar) 0 (L)
    gentime      (cPar) 0.0158106 (D)
end

[...]
```

It is possible that the format of the snapshot file will change to XML in future OMNeT++ releases.

6.9.6 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.

From the Feb99 release, OMNeT++ 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 OMNeT++ 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/omnetapp.h), which is currently 8K for Cmdenv and at least 16K for Tkenv.

> [The actual value is platform-dependent.]

`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.

6.10 Deriving new classes

6.10.1 cOwnedObject or not?

If you plan to implement a completely new class (as opposed to subclassing something already present in OMNeT++), you have to ask yourself whether you want the new class to be based on `cOwnedObject` or not. Note that we are *not* saying you should always subclass from `cOwnedObject`. Both solutions have advantages and disadvantages, which you have to consider individually for each class.

`cOwnedObject` already carries (or provides a framework for) significant functionality that is either relevant to your particular purpose or not. Subclassing `cOwnedObject` 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 OMNeT++ objects like `cQueue`, or you'll want to store OMNeT++ classes in your object, then you *must* subclass from `cOwnedObject`.

> [For simplicity, in the these sections ``OMNeT++ object'' should be understood as ``object of a class subclassed from `cOwnedObject`'']

The most significant features `cOwnedObject` has is the name string (which has to be stored somewhere, so it has its overhead) and ownership management (see section [6.11]) which also has the advantages but also some costs.

As a general rule, small `struct`-like classes like `IPAddress`, `MACAddress`, `RoutingTableEntry`, `TCPConnectionDescriptor`, etc. are better *not* sublassed from `cOwnedObject`. 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.

6.10.2 cOwnedObject virtual methods

Most classes in the simulation class library are descendants of `cOwnedObject`. If you want to derive a new class from `cOwnedObject` or a `cOwnedObject` descendant, you must redefine some member functions so that objects of the new type can fully co-operate with other parts of the simulation system. A more or less complete list of these functions is presented here. You do not need to worry about the length of the list: most functions are not absolutely necessary to implement. For example, you do not need to redefine `forEachChild()` unless your class is a container class.

The following methods **must** be implemented:

- *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 `NULL` as default name string.
- *Copy constructor*, which must have the following signature for a class `X`: `X(const X&)`. The copy constructor is used whenever an object is duplicated. The usual implementation of the copy constructor is to initialize the base class with the name (`getName()`) of the other object it receives, then call the assignment operator (see below).
- *Destructor*.
- *Duplication function,* `X *dup() const`. It should create and return an exact duplicate of the object. It is usually a one-line function, implemented with the help of the `new` operator and 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 your class contains other objects subclassed from `cOwnedObject`, either via pointers or as 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 data member; see the API Reference on `cOwnedObject` on how to implement `forEachChild()`. `forEachChild()` makes it possible for Tkenv to display the object

tree to you, to perform searches on it, etc. It is also used by `snapshot()` and some other library functions.

The following methods are **recommended** to implement:

- *Object info,* `std::string info()`. The `info()` function should return a one-line string describing the object's contents or state. `info()` is displayed at several places in Tkenv.
- *Detailed object info,* `std::string detailedInfo()`. This method may potentially be implemented in addition to `info()`; it can return a multi-line description. `detailedInfo()` is also displayed by Tkenv in the object's inspector.
- *Serialization*, `parsimPack()` and `parsimUnpack()` methods. These methods are needed for parallel simulation, if you want objects of this type to be transmitted across partitions.

6.10.3 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 [15])

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"));
```

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 [14]) to create blank objects to unmarshal into on the receiving side.

6.10.4 Details

We'll go through the details using an example. We create a new class `NewClass`, redefine all above mentioned `cOwnedObject` 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-`cOwnedObject` data (an array of `doubles`), an OMNeT++ object as data member (a `cQueue`), and a dynamically allocated OMNeT++ 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 cOwnedObject
{
  protected:
    int data;
    double *array;
    cQueue queue;
    cMessage *msg;
    ...
```

```
  public:
    NewClass(const char *name=NULL, 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 info();
};
```

We'll discuss the implementation method by method. Here's 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 d) : cOwnedObject(name)
{
    data = d;
    array = new double[10];
    take(&queue);
    msg = NULL;
}
```

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) : cOwnedObject(other.getName())
{
    array = new double[10];
    msg = NULL;
    take(&queue);
    operator=(other);
}
```

The copy constructor relies on the assignment operator. Because by convention the assignment operator does not copy the name member, it is passed here to the base class constructor. (Alternatively, we could have written `setName(other.getName())` into the function body.)

Note that pointer members have to be initialized (to `NULL` or to an allocated object/memory) before calling the assignment operator, to avoid crashes.

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 [6.11].

```
NewClass *NewClass::dup() const
{
    return new NewClass(*this);
}
```

The `dup()` functions is usually just one line, like the one above.

```
NewClass& NewClass::operator=(const NewClass& other)
{
    if (&other==this)
        return *this;
    cOwnedObject::operator=(other);

    data = other.data;

    for (int i=0; i<10; 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;
    return *this;
}
```

Complexity associated with copying and duplicating the object is concentrated in the assignment operator, so it is usually the one that requires the most work from you of all methods required by `cOwnedObject`.

If you do not want to implement object copying and duplication, you should implement the assignment operator to call `copyNotSupported()` -- it'll throw an exception that stops the simulation with an error message if this function is called.

The assignment operator copies contents of the `other` object to this one, except the name string. It should always return `*this`.

First, we should make sure we're not trying to copy the object to itself, because it might be disastrous. If so (that is, `&other==this`), we return immediately without doing anything.

The base class part is copied via invoking the assignment operator of the base class.

New data members are copied in the normal C++ way. If the class contains pointers, you'll 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 OMNeT++ 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 [6.11].

```
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 of `forEachChild()`.

```
std::string NewClass::info()
{
    std::stringstream out;
    out << "data=" << data << ", array[0]=" << array[0];
    return out.str();

}
```

The `info()` 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.

6.11 Object ownership management

6.11.1 The ownership tree

OMNeT++ has a built-in ownership management mechanism which is used for sanity checks, and as part of the infrastructure supporting Tkenv 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 `cOwnedObject`, 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 OMNeT++:

- enables methods like `getFullPath()` to be implemented.
- prevents certain types of programming errors, namely, those associated with wrong ownership handling.
- enables Tkenv 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 *must* 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: which object has the ownership of the message, why's that, and then probably you'll need to fix the logic somewhere in your program.

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 currently kept by) another module.

6.11.2 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-OMNeT++ 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's 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's 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 returning cOwnedObject* 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:

[Actual code in src/sim is structured somewhat differently, but the meaning is the same.]

```
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 cMessages, 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. The duplicate is supposed to 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.

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.

In fact, the same question arises in three places: the assignment operator operator=(), the copy constructor and the dup() method. In OMNeT++, the convention is that copying is implemented in the assignment operator, and the other two just rely on it. (The copy constructor just constructs an empty object and invokes assignment, while dup() is implemented as new cArray(*this)).

---

7 Building Simulation Programs

7.1 Overview

As it was already mentioned, an OMNeT++ model physically consists of the following parts:

- NED language topology description(s). These are files with the `.ned` extension.
- Message definitions, in files with `.msg` extension.
- Simple modules implementations and other C++ code, in `.cc` files (or `.cpp`, on Windows)

To build an executable simulation program, you first need to translate the MSG files into C++, using the message compiler (`opp_msgc`). After this step, the process is the same as building any C/C++ program from source: all C++ sources need to be compiled into object files (`.o` files (using gcc on Mac, Linux) or mingw on Windows) and all object files need to be linked with the necessary libraries to get an executable or shared library.

> **NOTE**
> Compiling NED files directly to C++ classes is no longer supported in OMNeT++ 4.0. NED files are always dynamically loaded.

File names for libraries differ for Unix/Linux and for Windows, and also different for static and shared libraries. Let us suppose you have a library called Tkenv. If you are compiling with gcc or mingw, the file name for the static library would be something like `libopptkenv[d].a`, and the shared library would be called `libopptkenv[d].so`. (libopptkenvd.so would be used for the debug version while libopptkenv.so is for the release build.)

> **NOTE**
> On Windows, shared libraries have the `.dll` extension instead of `.so` On Mac OS X, shared libraries have the `.dylib` extension instead of `.so`.

In OMNeT++ 4.0 we recommend to use shared libraries whenever it is possible. You'll need to link with the following libraries:

- The simulation kernel and class library, called *oppsim* (file `liboppsim.[so|dll|dylib]` etc).
- User interfaces. The common part of all user interfaces is the *oppenvir* library (file `liboppenvir.[so|dll|dylib]`, etc), and the specific user interfaces are *opptkenv* and *oppcmdenv* (`libopptkenv.[so|dll|dylib]`, `liboppcmdenv.[so|dll|dylib]`, etc). You have to link with *oppenvir*, plus *opptkenv* or *oppcmdenv* or both.

Luckily, you do not have to worry about the above details, because automatic tools like `opp_makemake` will take care of the hard part for you.

The following figure gives an overview of the process of building and running simulation programs.

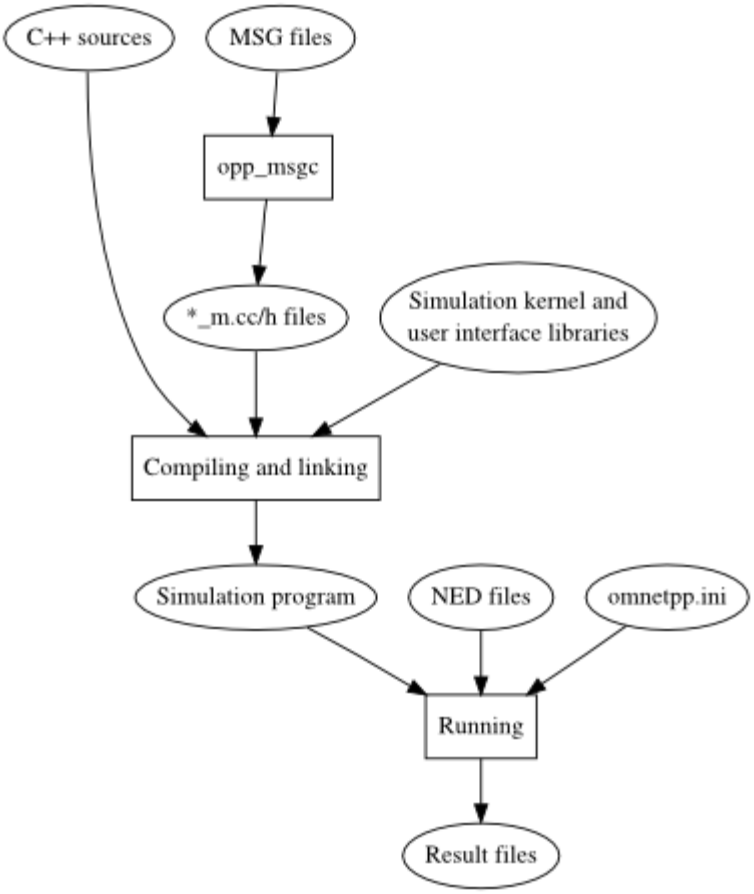


Figure: Building and running simulation

This section discusses how to use the simulation system on the following platforms:

- Unix (Linux/Mac OS X) with gcc
- Windows with the included MinGW compiler

7.2 Using gcc

The following section applies to using OMNeT++ on Linux, Solaris, Mac OS X, FreeBSD and other Unix derivatives, and also to MinGW on Windows.

> **NOTE**
> The `doc/` directory of your OMNeT++ installation contains `Readme.`*<platform>* files that provide more detailed platform specific instructions.

7.2.1 The opp_makemake tool

The `opp_makemake` tool can automatically generate a `Makefile` for your simulation program, based on the source files in the current directory or directory tree. `opp_makemake` has several options; `opp_makemake -h` displays help.

The most important options 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, -Xdirectory, --except directory` : With `-r` and `--deep` option: ignore the given directory.
- `-dsubdir, -d subdir, --subdir subdir` : Causes make to recursively descend into the given directory.
- `-n, --nolink` : Produce object files but do not create executable or library.
- `-s, --make-so` : Build shared library (.so, .dll or .dylib).
- `-a, --make-lib` : Create static library (.a or .lib).
- `-Idir` : Additional NED and C++ include directory.
- `-Ldir` : Add a directory to the library path.
- `-llibrary` : Additional library to link against.

7.2.2 Basic use

Once you have the source files (`*.ned`, `*.msg`, `*.cc`, `*.h`) in a directory, change there and type:

```
$ opp_makemake
```

This will create a file named `Makefile`. If you type `make`, your simulation program should build.

If you already had a `Makefile` in that directory, `opp_makemake` will refuse to overwrite it. You can force overwriting the old `Makefile` with the -f option:

```
$ opp_makemake -f
```

The name of the output file will be derived from the name of the project directory (see later). You can override it with the `-o` option:

```
$ opp_makemake -f -o aloha
```

In addition to the default target that builds the simulation executable, the `Makefile` also contains the following targets:

Target	Action
all	The default target is to build the simulation executable
depend	Adds (or refreshes) dependencies in the `Makefile`
clean	Deletes all files that were produced by the make process

7.2.3 Debug and release builds

`opp_makemake` generates a makefile that can create both release and debug builds. By default it creates debug version, but it is easy to override this behaviour. Just define the `MODE` variable on the `make` command line.

```
$ make MODE=release
```

If you want to create release builds by default you should use the `--mode` *mode* option for `opp_makemake` when generating your makefiles.

```
$ opp_makemake --mode release ...
```

7.2.4 Using external C/C++ libraries

If you are using external libraries you should specify the include path for the header files with the `-I` *includedir* option. You should specify this option if you are using anything outside from the source directory tree (except the system and OMNeT++ headers which are always included automatically)

To define an external library to be linked with, use `-Ldir` to specify the directory of the external library and `-llibrary` to specify the name of the external dependency.

7.2.5 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 turns 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. If you need 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
```

7.2.6 Automatic include dirs

If your source tree contains several subdirectories (maybe several levels deep), it can be annoying that you should specify relative paths for your header files in your `.cc` files or you should specify the include path explicitly by the `-I includepath` option. `opp_makemake` has a command line option, which adds all directories in the current source tree to the compiler command line. This option is turned on by default.

> **NOTE**
> You may turn off this mechanism with the `--no-deep-includes` option.

The only requirement is that your `#include` statements must unambigously specify the name of the header file. (i.e. if you have two `common.h` files, one in subdir1 and the other in subdir2 specify `#include "subdir1/common.h"` instead of `#include "common.h"`. If you want to include a directory which is outside of your source directory tree you always must specify it with the `-I external/include/dir` option.

7.2.7 Dependency handling

Dependency information is used by the makefile to minimize the time required to compile and link your project. If your makefile contains up-to date dependency info -- only files changed since you last compiled your project will be re-compiled or linked.

`opp_makemake` automatically adds dependencies to the makefile. You can regenerate the dependencies by typing `make depend` any time. The warnings during the dependency generation process can be safely ignored.

You may generate and add dependencies to the `makefile` manually using the `opp_makedep` tool. Use `opp_makedep --help` to display the supported command line options.

> **NOTE**
> The dependency generator does not handle conditional macros and includes. Conditionally included header files are always added to the file's dependency list.

7.2.8 Out-of-directory build

The build system creates object and executable files in a separate directory, called *output directory*. The structure of the output folder will be the same as your sourcefolder structure except that it will be placed in the `out/configname` directory. The `configname` part will mirror your compiler toolchain and build mode settings. (i.e. The result of a debug build with gcc will be placed in `out/gcc-debug`)

The location of the generated output file is determined by the -O option. (The default value is 'out', relative to the project root directory):

```
$ opp_makemake -O ../tmp/obj
```

> **NOTE**
> The project directory is the first ancestor of the current directory which contains a `.project` file).

> **NOTE**
> Source files (i.e. those created by the `opp_msgc` compiler) will be generated in the source folder rather than in the output folder.

7.2.9 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 avoids the linking step, which is useful for top-level makefiles that only invoke other makefiles, or if you want to do the linking manually.

7.2.10 Recursive builds

The `--recurse` option enables recursive make: when you build the simulation, make will descend into the subdirectories and runs make in them too. By default, `--recurse` decends into all subdirectories; the -X directory 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 OMNeT++ 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 [7.2.11])

> **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

7.2.11 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 `makefrag` 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 wants to override the default target in the makefile.

7.2.12 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.

7.2.13 A multi-directory example

For a complex example of using `opp_makemake`, we will check how to create the makefiles for the mobility-framework. First take a look at the project's directory structure and find the directories that should be used as source trees:

```
mobility-framework
    bitmaps
    contrib <-- source tree (build libmfcontrib.so from this dir)
    core <-- source tree (build libmfcore.so from this dir)
    docs
    network
    template
    testSuite <-- 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` of course).

First create the makefile for the core directory (build a shared lib from all .cc files found in the core subtree and name it 'mfcore'):

```
$ cd core && opp_makemake -f --deep --make-so -o mfcore -O out
```

The contrib directory is depending on `mfcore` so we use the -L and -l options to specify the library we should link with. Note that we must also add the include directories manually from the core source tree, because autodiscovery works only in the same source tree:

```
$ cd contrib && opp_makemake -f --deep --make-so -o mfcontrib -O out \\
  -I../core/basicModules -I../core/utils -L../out/$(CONFIGNAME)/core -lmfcore
```

The testSuite will be created as an executable file which depends on both `mfcontrib` and `mfcore`.

```
$ cd testSuite && opp_makemake -f --deep -o testSuite -O out
    -I../core/utils -I../core/basicModules -I../contrib/utils \\
    -I../contrib/applLayer -L../out/$(CONFIGNAME)/contrib -lmfcontrib
```

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 (-d dirname).

```
$ opp_makemake -f --nolink -O out -d testSuite -d core -d contrib -X.
```

Finally we have to specify the dependencies between the above directories. Add the lines below to the `makefrag` file in the project directory root.

```
contrib_dir: core_dir
testSuite_dir: contrib_dir
```

---

8 Configuring Simulations

8.1 Configuring simulations

Configuration and input data for the simulation are in a configuration file usually called `omnetpp.ini`.

The following sections explain `omnetpp.ini`.

8.2 The configuration file: omnetpp.ini

8.2.1 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*.

Lines that start with ``#'' are comments.

8.2.2 File syntax

The ini file is a text file consisting of entries grouped into different sections. The order of the sections doesn't matter.

Lines that start with "#" are comments, and will be ignored during processing.

Long lines can be broken up using the backslash notation: if the last character of a line is "\", it will be merged with the next line.

There is no limit on the file size or the maximum line length.

Example:

```
[General]
# this is a comment
**.foo = "this is a single value \
for the foo parameter"
```

8.2.3 File inclusion

OMNeT++ supports including an ini file in another, via the `include` keyword. This feature allows you to partition large ini files into logical units, fixed and varying part etc.

An example:

```
# omnetpp.ini
...
include parameters.ini
include per-run-pars.ini
...
```

You 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.)

8.3 Sections

8.3.1 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).

It is important to note, that the loaded NED files may contain any number of modules, channel and *any number of networks* as well. It does not matter whether you use all or just some of them in the simulations. You will be able to select `any` of the networks that occur in the loaded NED files using the `network= omnetpp.ini` entry, and as long as every module, channel etc for it has been loaded, network setup will be successful.

8.3.2 Named configurations

Named configurations are sections of the form `[Config <configname>]`, 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 keys (such as user interface selection) are only accepted in the `[General]` section, but most of them can go into Config sections as well.

When you run a simulation, you need 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. (Tkenv, of course, lets you select the configuration from a dialog.)

8.3.3 Section inheritance

Actually, when you activate the PureAloha configuration, the contents of the `[General]` section will also be taken into account: if some configuration key 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 a fall-back section explicitly, using the extends= key. Consider the following ini file skeleton:

```
[General]
...
[Config SlottedAlohaBase]
...
[Config SlottedAloha1]
extends = SlottedAlohaBase
...
[Config SlottedAloha2]
extends = SlottedAlohaBase
...
[Config SlottedAloha2a]
extends = SlottedAloha2
...
[Config SlottedAloha2b]
extends = SlottedAloha2
...
```

If you activate the `SlottedAloha2b` configuration, lookups will consider sections in the following order (this is also called the *section fallback chain*): `SlottedAloha2b`, `SlottedAloha2`, `SlottedAlohaBase`, `General`.

The effect is the same as if the contents of the sections SlottedAloha2b, SlottedAloha2, SlottedAlohaBase 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 concept is similar to inheritance in object-oriented languages, and benefits are similar too: you can factor out the common parts of several configurations into a "base" configuration, and the other way round, you can reuse existing configurations (as opposed to copying them) by using them as a base. In practice you 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.

If you are experimenting a lot with different parameter settings of a simulation model, these techniques will make it a lot easier to manage ini files.

8.4 Setting 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 or 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 [6.1.5]).

An example `omnetpp.ini` which sets the `numHosts` parameter of the toplevel module and the `transactionsPerSecond` parameter of the `server` module:

```
[General]
net.numHosts = 15
net.server.transactionsPerSecond = 100
```

8.4.1 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`. OMNeT++ supports *wildcards patterns* which allow for setting several model parameters at once.

The notation is a variation on the usual glob-style patterns. The most apparent differences to the usual 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 `{a-zA-Z}` not `[a-zA-Z]`, because square brackets are already 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 character (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 (e.g. `99`)
- `[38..150]` : *index range*: any number in square brackets in the range 38..150 (e.g. `[99]`)
- backslash (\) : takes away the special meaning of the subsequent character

Precedence

If you use wildcards, the order of entries is important: if a parameter name matches several wildcards-patterns, the *first* matching occurrence is used. This means that you need 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}`. If you want 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 literal backslash (and NOT as 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
```

8.4.2 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 the 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 you must explicitly enable the interactive mode with the `--cmdenv-interactive=true` option otherwise you will get an error when running the simulation.

More precisely, parameter resolution takes place as follows:

- If the parameter is assigned in NED, it cannot be overridden in the configuration. The value is applied and the process finishes.
- If the first match is a value line (matches *<parameter-fullpath>*`=somevalue`), the value is applied and the process finishes.
- If the first match is a *<parameter-fullpath>*`=default` line, the default value is applied and the process

finishes.

- If the first match is a *<parameter-fullpath>*=ask line, the parameter will be asked from the user interactively (UI dependent).
- If there was no match and the parameter has a default value, it is applied and the process finishes.
- 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.

8.5 Parameter studies

8.5.1 Basic use

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. OMNeT++ 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, and generated a (partial) ini file and run the simulation program in each iteration.

OMNeT++ 4.0 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's 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 does it get run? First of all, Cmdenv with the '-x' option will tell you how many simulation runs a given section expands to. (You'll of course use Cmdenv for batch runs, not Tkenv.)

```
$ aloha -u Cmdenv -x AlohaStudy

`\opp` Discrete Event Simulation
...
Config: AlohaStudy
Number of runs: 24
```

If you add the '-g' option, the program will also print out the values of the iteration variables for each run. (Use '-G' for 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 -u Cmdenv -x AlohaStudy -g
`\opp` 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 OMNeT++ 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 OMNeT++.

Actually, it is also possible to get Cmdenv execute all runs in one go, by simply omitting the '-r' option.

```
$ aloha -u Cmdenv -c AlohaStudy

`\opp` 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 is terminated -- which may not be what the user wants).

Let us get back to the ini file. We had:

```
[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 gets textually replaced with the current iteration value. The values to iterate over do not need to be numbers (unless you want to use the *"a..b"* or *"a..b step c"* syntax), 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 ${..} inside a string constant, quote "{" with a backslash, and write "$\{..}".

8.5.2 Named iteration variables

You can assign names to iteration variables, which has the advantage that you'll see meaningful names instead of $0 and $1 in the Cmdenv output, and also lets you refer to the variables at more than one place 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=).

There are also 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, it gets assigned a Run ID, which uniquely identifies that instance of running the simulation: if you run the same thing again, it will get a different Run ID. Run ID is a 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.

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 key. An example:

```
*.numNodes = ${n=10..100 step 10}
**.numNeighbors = ${m=2..10 step 2}
constraint = $m <= sqrt($n)
```

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**
> It is not supported to refer to other iteration variables in the definition of an iteration variable (i.e. you cannot write things like ${j=$i..10}), although it might get implemented in future OMNeT++ releases.

8.5.3 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 keys related to this: repeat= and seed-set=. The first one simple 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 of 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 OMNeT++ 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 makes 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, you can just 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's no need for an extra loop.

It is of course also possible to manually specify individual seeds for simulations. This is rarely necessary, but we can use it here to demonstrate another feature, parallel iterators:

```
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.

8.6 Parameter Studies and Result Analysis

8.6.1 Output vectors and scalars

In OMNeT++ 3.x, the default result file names were ``omnetpp.vec'' and ``omnetpp.sca''. This is not very convenient for batch execution, where an output vector file created in one run would be overwritten in the next run. Thus, we have changed the default file names to make them differ for every run. The new defaults are:

```
output-vector-file = "${resultdir}/${configname}-${runnumber}.vec"
output-scalar-file = "${resultdir}/${configname}-${runnumber}.sca"
```

Where `${resultdir}` is the value of the `result-dir` configuration option, and defaults to `results/`. So the

above defaults generate file names like "results/PureAloha-0.vec", "results/PureAloha-1.vec", and so on.

Also, in OMNeT++ 3.x output scalar files were always appended to by the simulation program, rather than being overwritten. This behavior was changed in 4.0 to make it consistent with vector files, that is, output scalar files are also overwritten by the simulator, and not appended to.

> **NOTE**
> The old behavior can be turned back on by setting output-scalar-file-append=true.

Although it has nothing to do with our main topic (ini files), this is a good place to mention that the format of result files have been extended to include meta info such as the run ID, network name, all configuration settings, etc. These data make the files more self-documenting, which can be valuable during the result analysis phase, and increase reproducibility of the results. Another change is that vector data are now recorded into the file clustered by the output vectors, which (combined with index files) allows much more efficient processing.

8.6.2 Configuring output vectors

As a simulation program is evolving, it is becoming capable of collecting more and more statistics. The size of output vector files can easily reach a magnitude of several ten or hundred megabytes, but very often, only some of the recorded statistics are interesting to the analyst.

In OMNeT++, you can control how `cOutVector` objects record data to disk. You can turn output vectors on/off or you can assign a result collection interval. By default, all output vectors are turned on.

> **NOTE**
> The way of configuring output vectors has changed. In OMNeT++ 3.x, the keys for enabling-disabling a vector and specifying recording interval were <module-and-vectorname-pattern>.enabled, and <module-and-vectorname-pattern>.interval. The "enabled" and "interval" words changed to "vector-recording" and "vector-recording-interval". The reason for this change is that per-object configuration keys are now required to have a hyphen in their names, to make it possible to tell them apart from module parameter assignments. This allows the simulator to catch mistyped config keys in ini files. The syntax for specifying the recording interval has also been extended (in a backward compatible way) to accept multiple intervals, separated by comma.

Output vectors can be configured with the following syntax:

```
module-pathname.objectname.vector-recording = true/false
module-pathname.objectname.vector-recording-interval = start1..stop1,
                                                       start2..stop2, ...
```

Either *start* or *stop* can be omitted, to mean the beginning and the end of the simulation, respectively.

The object name is the string passed to `cOutVector` in its constructor or with the `setName()` member function.

```
cOutVector eed("End-to-End Delay");
```

Start and stop values can be any time specification accepted in NED and config files (e.g. *10h 30m 45.2s*).

As with parameter names, wildcards are allowed in the object names and module path names.

An example:

```
[General]
**.vector-recording-interval = 1s..60s
**.End-to-End Delay.vector-recording = true
**.Router2.**.vector-recording = true
**.vector-recording = false
```

The above configuration limits collection of all output vectors to the 1s..60s interval, and disables collection of output vectors except all end-to-end delays and the ones in any module called Router2.

8.6.3 Saving parameters as scalars

When you are running several simulations with different parameter settings, you'll 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, OMNeT++ 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:

```
# NEVER USE:
**.host[*].generationInterval = exponential( ${mean=0.2s, 0.4s, 0.6s} )
# INSTEAD: Specify the unit outside of the variable
**.host[*].generationInterval = exponential( ${mean=0.2, 0.4, 0.6}s )
```

Module parameters can also be saved, but this has to be requested by the user, by configuring save-as-scalar=true for the parameters in question. The configuration key is a pattern that identifies the parameter, plus ".save-as-scalar". An example:

```
**.host[*].networkLoad.save-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.save-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.save-as-scalar = true  # wrong: saves random values!
```

These pitfalls are not rare in practice, so it is usually more convenient 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. save-as-scalar=true is not needed because iteration variables are automatically saved into the result files.

8.6.4 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 person 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 parameter settings. 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.

OMNeT++ result analysis tools can take advantage of experiment/measurement/replication labels recorded into result files, and organize simulation runs and recorded output scalars and vectors accordingly on the user interface.

These labels can be explicitly specified in the ini file using the experiment=, measurement= and replication= config keys. If they are missing, the default is the following:

```
experiment = "${configname}"
measurement = "${iterationvars}"
replication = "#${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 for the seed-set. Thus, for our first example the experiment-measurement-replication tree would look like this:

```
"PureAloha" Unknown LaTeX command  \textrm --experiment
  $N=1,$mean=0.2 Unknown LaTeX command  \textrm  -- measurement
    #0, seed-set=0 Unknown LaTeX command  \textrm  -- 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.

We believe that the default way of generating experiment-measurement-replication labels will be useful and sufficient in the majority of the simulation studies. However, you can customize it if needed. For example, here is a way to join two configurations into one experiment:

```
[Config PureAloha_Part1]
experiment = "PureAloha"
...
[Config PureAloha_Part2]
experiment = "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.

8.7 Configuring the random number generators

The random number architecture of OMNeT++ was already outlined in section [6.4]. Here we'll cover the configuration of RNGs in `omnetpp.ini`.

8.7.1 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 [6.4]). Referencing an RNG number greater or equal to this number (from a simple module or NED file) will cause a runtime error.

8.7.2 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 OMNeT++ 2.3 and earlier versions, with a cycle length of $2^{31}-2$), and `"cAkaroaRNG"` (Akaroa's random number generator, see section [9.5]).

8.7.3 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
```

This mapping allows variance reduction techniques to be applied to OMNeT++ models, without any model change or recompilation.

8.7.4 Automatic seed selection

Automatic seed selection gets used for an RNG if you don't 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, OMNeT++ always selects the same seed value for any simulation model. If the run number or the RNG number is different, OMNeT++ does its best to choose different seeds which are also sufficiently apart 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=` entry) 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).

> [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.]

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, OMNeT++ 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.

8.7.5 Manual seed configuration

In some cases you may want manually configure seed values. Reasons for doing that may be that you want to use variance reduction techniques, or you may want to use 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`, and OMNeT++ provides a standalone program called `opp_lcg32_seedtool` to generate good seed values that are sufficiently apart in the RNG's sequence.

---

9 Running Simulations

9.1 Introduction

This chapter describes how to run simulations. It covers basic usage, user interfaces, batch runs, how to use Akaroa, and also explains how to solve the most common errors.

9.1.1 Running a simulation executable

By default, the output of an `opp_makemake`-generated makefile is a simulation executable that can be run directly. In simple cases, this executable can be run without command-line arguments, but usually one will need to specify options to specify what ini file to use, which user interface to activate, where to load NED files from, and so on.

Getting help

The following sections describe the most frequently used command-line options. To get a complete list of supported command line options, run a simulation executable (or `opp_run`) with the `-h` option.

```
$ ./fifo -h
```

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 get merged. This allows for partitioning the configuration into separate files, for example to simulation options, module parameters and result recording options.

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:

- OMNeT++ checks for NED path specified on the command line with the `-n` option
- if not found on the command line, it checks for the NEDPATH environment variable
- the **`ned-path`** entry from the ini file gets appended to the result of the above steps
- if the result is still empty, it falls back to "." (the current directory)

Selecting a user interface

OMNeT++ simulations can be run under different user interfaces. Currently the following user interfaces are supported:

- Tkenv: Tcl/Tk-based graphical, windowing user interface
- Cmdenv: command-line user interface for batch execution

You would typically test and debug your simulation under Tkenv, then run actual simulation experiments from the command line or shell script, using Cmdenv. Tkenv is also better suited for educational or demonstration purposes.

Both Tkenv and Cmdenv are provided in the form of a library, and you may choose between them by linking one or both into your simulation executable. (Creating the executable was described in chapter [7]). Both user interfaces are supported on Unix and Windows platforms.

You can choose which runtime environment is included in your simulation executable when you generate your makefile. By default both Tkenv and Cmdenv is linked in so you can choose between them during runtime, but it is possible to specify only a single user interface with the `-u Cmdenv` or `-u Tkenv` option on the `opp_makemake` command line. This can be useful if you intend to run your simulations on a machine where Tcl/Tk is not installed.

By default, Tkenv will be used if both runtime environment is present in your executable, but you can override this with the `user-interface=Cmdenv` in your ini file or by specifying `-u Cmdenv` on the command line. If both the config option and the command line option are present, the command line option takes precedence.

Selecting a configuration and run number

Configurations can be selected with the `-c <configname>` command line option. If you do not specify the configuration and you are running under:

- Tkenv: the runtime environment will prompt you to choose one.
- Cmdenv: the `General` configuration will be executed.

User interfaces may support the `-r runnumber` option to select runs, either one or more, depending on the type of the user interface.

There are several command line options to get information about the iteration variables and the number of runs in the configurations:

- `-a` -- Prints all configuration names and the number of runs in them.
- `-x <configname>` -- Prints the number of runs available in the given configuration.
- `-g` -- Prints the unrolled configuration (together with the -x option) and expands the iteration variables.

`-G` -- Prints even more details than -g.

Loading extra libraries

OMNeT++ allows you to load shared libraries at runtime. This means that you can create simulation models as a shared library and load the model later into a different executable without the need to explicitly link against that library. This approach has several advantages.

- It is possible to distribute the model as a shared library. Others may be able to use it without recompiling it.
- You can split a large model into smaller, reusable components.
- You can mix several models (even from different projects) without the need of linking or compiling.

Use the `-l libraryname` command line option to load a library dynamically at run time. OMNeT++ will attempt to load it using the `dlopen()` or `LoadLibrary()` functions and automatically registers all simple modules in the library.

The prefix and suffix from the library name can be omitted (the extension (`.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 (name.dll, libname.dll, libname.so and libname.dylib can be loaded with `-l name`).

It is also possible to specify the libraries to be loaded in the ini file with the `load-libs=` config option. The values from the command line and the config file will be merged.

> **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.

> **NOTE**
> You must ensure that the library can be accessed by OMNeT++ . You have to specify the path with your library name (pre- and postfixes of the library filename still can be omitted) or adjust your shared library path according to your OS. (On Windows set the PATH, on Unix set LD_LIBRARY_PATH and on Mac OS X set the DYLD_LIBRARY_PATH as needed.)

9.1.2 Running a shared library

Shared libraries can be run using the `opp_run` program. Both `opp_run` and simulation executables are capable of loading additional shared libraries; actually, `opp_run` is nothing else than an empty simulation executable.

Example:

```
opp_run -l mymodel
```

The above example will load the model found in `libmymodel.so` and execute it.

9.1.3 Controlling the run

There are several useful configuration options that control how a simulation is run.

- **`cmdenv-express-mode`** -- Provides only minimal status updates on the console.
- **`cmdenv-interactive`** -- Allows asking interactively for missing parameter values.
- **`cmdenv-status-frequency`** -- How often the status is written to the console.
- **`cpu-time-limit`** -- Limit how long the simulation should run (in wall clock time)
- **`sim-time-limit`** -- Limit how long the simulation should run (in simulation time)
- **`debug-on-errors`** -- If the runtime detects any error it will generate a breakpoint so you will be able to check the location and the context of the problem in your debugger.
- **`fingerprint`** -- The simulation kernel computes a checksum while running the simulation. It is calculated

from the module id and from the current simulation time of each event. If you specify the **`fingerprint`** option in the config file, the simulation runtime will compare the computed checksum with the provided one. If there is a difference it will generate an error. This feature is very useful if you make some cosmetic changes to your source and want to be reasonable sure that your changes did not altered the behaviour f the model.

- **`record-eventlog`** -- You can turn on the recording of the simulator events. The resulting file can be analyzed later in the IDE with the sequence chart tool.

NOTE
It is possible to specify a configuration option also on command line (in which case the command line takes precedence). You should prefix the option name with a double dash (--) and should not put any spaces around the equal sign (e.g. `--debug-on-errors=true`)

To get the list of all possible configuration options type:

```
opp_run -h config
```

9.2 Cmdenv: the command-line interface

The command line user interface is a small, portable and fast 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. If one run stops with an error message, subsequent ones will still be executed. The runs to be executed can be passed via command-line argument or in the ini file.

9.2.1 Example run

When run the Fifo example under Cmdenv, you should see something like this:

```
$ ./fifo -u Cmdenv -c Fifo1

OMNeT++ Discrete Event Simulation  (C) 1992-2008 Andras Varga, OpenSim Ltd.
Version: 4.0, 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%
completed
     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, periodically it 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 output from them are displayed. On my machine this run took 34 seconds. This Cmdenv output can be customized via `omnetpp.ini` entries. The output file `results/Fifo1-0.vec` contains vector data recorded during simulation (here, queueing times), and it can be processed using the IDE or other tools.

9.2.2 Command-line switches

The command line environment allows you to specify more than one run by using the `-r 2,4,6..8` format. See [9.4] for more information about running simulation batches.

9.2.3 Cmdenv ini file options

`cmdenv-runs-to-execute` specifies which simulation runs should be executed. It accepts 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 that have ini file sections; if no runs are specified in the ini file, Cmdenv does one run. The -r command line option overrides this ini file setting.

Cmdenv can be executed in two modes, selected by the **`cmdenv-express-mode`** ini file option:

- **Normal** (non-express) mode is for debugging: detailed information will be written to the standard output (event banners, module output, etc).
- **Express** mode can be used for long simulation runs: only periodical status update is displayed about the progress of the simulation.

`cmdenv-performance-display` affects express mode only: it controls whether to print 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.

For a full list of options, see the ones beginning with `cmdenv` in Appendix [23].

9.2.4 Interpreting Cmdenv output

When the simulation is running in ``express'' mode with detailed performance display enabled, Cmdenv periodically outputs a three-line status info 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 info about simulation performance:

- `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 independent of the size (number of modules) in your model.
- `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 cell-level ATM simulation you'll have very hight values (10^9), while in a bank teller 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 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.

9.3 Tkenv: the graphical user interface

Tkenv is a portable graphical windowing user interface. Tkenv supports interactive execution of the simulation, tracing and debugging. Tkenv is recommended in the development stage of a simulation or for presentation and educational 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.

9.3.1 Command-line switches

A simulation program built with Tkenv accepts all the general command line switches. In addition the `-r runnumber` option allows only specifying a single run. Multiple runs are not supported.

Tkenv configuration options:

- **`tkenv-default-config`**: Specifies which config Tkenv should set up automatically on startup. The default is to ask the user.
- **`tkenv-default-run`**: Specifies which run (of the default config, see tkenv-default-config) Tkenv should set up automatically on startup. The default is to ask the user.
- **`tkenv-extra-stack`**: Specifies the extra amount of stack that is reserved for each activity() simple module when the simulation is run under Tkenv.
- **`tkenv-image-path`**: Specifies the path for loading module icons.
- **`tkenv-plugin-path`**: Specifies the search path for Tkenv plugins. Tkenv plugins are .tcl files that get evaluated on startup.

Tkenv specific configuration options can be specified also on command line by prefixing them with double dash (e.g --tkenv-option=value). See Appendix [23] for the list of possible configuration options.

> **NOTE**
> The usage of the Tkenv user interface is detailed in the OMNeT++ User Guide's Tkenv chapter.

9.4 Batch execution

Once your model works reliably, you'll usually want to run several simulations. You may want to run the model with various parameter settings, or you may want *(should want?)* to run the same model with the same parameter settings but with different random number generator seeds, to achieve statistically more reliable results.

Running a simulation several times by hand can easily become tedious, and then a good solution is to write a control script that takes care of the task automatically. Unix shell is a natural language choice to write the control script in, but other languages like Perl, Matlab/Octave, Tcl, Ruby might also have justification for this purpose.

Of course before running simulation batches you must set a condition to stop your simulation. This is usually a time limit set by the **`sim-time-limit`** configuration option, but you can limit your simulation by using wall clock time (**`cpu-time-limit`**) or by directly ending a simulation with an API call if some condition is true.

9.4.1 Using Cmdenv

To execute more than one run using the Cmdenv use the `-r` option and specify the runs in a comma separated format 1,2,4,9..11 or you may leave out the `-r` option to execute all runs in the experiment.

> **WARNING**
> Although it is very convenient, we do not recommended to use this method for running simulation batches. Specifying more than one runnumber would run these simulations in the same process. This method is more prone to C++ programming errors. A failure in a single run may stop and kill all the runs following it. If you want to execute more than one run we recommend to run each of them in a separate process. Use the `opp_runall` for this purpose.

9.4.2 Using shell scripts

The following script executes a simulation named `wireless` several times, with parameters for the different runs given in the `runs.ini` file.

Before you are executing your simulation batch, you may check how many runs are available in the configuration you are using. Use the `-x config` command line option to print the number of runs or add the `-g` to get more details.

```
#! /bin/sh
./wireless -f runs.ini -r 1
./wireless -f runs.ini -r 2
./wireless -f runs.ini -r 3
./wireless -f runs.ini -r 4
...
./wireless -f runs.ini -r 10
```

To run the above script, type it in a text file called e.g. `run`, give it `x` (executable) permission using `chmod`, then you can execute it by typing `./run`:

```
$ chmod +x run
$ ./run
```

You can simplify the above script by using a *for* loop. In the example below, the variable `i` iterates through the values of list given after the `in` keyword. It is very practical, since you can leave out or add runs, or change the order of runs by simply editing the list -- to demonstrate this, we skip run 6, and include run 15 instead.

```
#! /bin/sh
for i in 3 2 1 4 5 7 15 8 9 10; do
   ./wireless -f runs.ini -r $i
done
```

If you have many runs, you can use a C-style loop:

```
#! /bin/sh
for ((i=1; $i<50; i++)); do
   ./wireless -f runs.ini -r $i
done
```

9.4.3 Using opp_runall

OMNeT++ has a utility program called `opp_runall` which allows you to execute a simulation batch in command line mode. You must specify the whole command line you would use to run your batch in Cmdenv. There are advantages running your batches this way:

- Each simulation run executes in a separate operating system process. This means that a crash because of a programming error does not affect the outcome of the other runs. They are totally independent of each other.
- If you happen to have a multi core/processor machine, you can take advantage of the processing power by running sevaral runs parallel.

The command basically creates a makefile which contains a separate target for each run. By default the makefile will be executed causing each target to run. You can give additional options to the `opp_runall` command to activate parallel building. The `-j` option can be used to specify the maximum number of parallel runs allowed.

> **WARNING**
> Use the parallel execution option only if you have enough memory to run several simulations side by side. If you run out of memory your operating system will start swapping and the overall performance of the system will be greatly reduced. Always specify the number of processes after the `-j` option otherwise the `make` program will try to start *all* runs at the same time. As a rule of thumb: if you have 4 cores (and enough memory), use `-j4`.

The form of the command is:

```
opp_runall -j2 ./aloha -u Cmdenv -c PureAlohaExperiment -r 0..23
```

You can use the `-x ConfigName -g` command line options with your simulation to check the number of available runs.

Using the `--export filename` option only generates the `makefile`, but does not start it. You can run your batch later by invoking the generated makefile.

9.5 Akaroa support: Multiple Replications in Parallel

9.5.1 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 is the initial transient over, when can we start collecting data? We usually do not 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'', can reach the required sample size to be statistically trustable.

Neither questions are 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 [Pawlikowsky02] 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 runtime.

9.5.2 What is Akaroa

Akaroa [Akaroa99] 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.

9.5.3 Using Akaroa with OMNeT++

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 help you set up and successfully use Akaroa -- for that you need to read the Akaroa manual.

Configuring OMNeT++ for Akaroa

First of all, you have to compile OMNeT++ with Akaroa support enabled.

The OMNeT++ 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 [6]). 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.

> [For more details on the plugin mechanism these settings make use of, see section .]

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. It is vital to obtain random numbers from Akaroa: otherwise, all simulation processes would 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, all output vectors are under Akaroa control; the

```
<modulename>.<vectorname>.with-akaroa = false
```

setting can be used to make Akaroa ignore specific vectors. You can use the `*`, `**` wildcards here (see section [8.4.1]). For example, if you only want a few vectors be placed under Akaroa, you can use the following trick:

```
<modulename>.<vectorname1>.with-akaroa = true
<modulename>.<vectorname2>.with-akaroa = true
```

```
...
**.*.with-akaroa = false  # catches everything not matched above
```

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 OMNeT++ simulation processes run with the same settings, they would overwrite each other's output files (e.g. `omnetpp.vec`, `omnetpp.sca`). 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).

9.6 Troubleshooting

9.6.1 Unrecognized configuration option

If you receive an error message about unrecognized configuration options you may use `-h config` or `-h configdetails` options to display all possible configuration options and their descriptions.

9.6.2 Stack problems

``Stack violation (*FooModule* stack too small?) in module *bar.foo*''

OMNeT++ detected that the module has used more stack space than it has allocated. The solution is to increase the stack for that module type. You can call the `getStackUsage()` from `finish()` to find out actually how much stack the module used.

``Error: Cannot allocate *nn* bytes stack for module *foo.bar*''

The resolution depends on whether you are using OMNeT++ on Unix or on Windows.

Unix. If you get the above message, you have to increase the total stack size (the sum of all coroutine stacks). You can do so in `omnetpp.ini`:

```
[General]
total-stack-kb = 2048 # 2MB
```

There is no penalty if you set `total-stack-kb` too high. I recommend to set it to a few K less than the maximum process stack size allowed by the operating system (`ulimit -s`; see next section).

Windows. You need to set a *low* (!) ``reserved stack size'' in the linker options, for example 64K (/stack:65536 linker flag) will do. The ``reserved stack size'' is an attribute in the Windows exe files' internal header. It can be set from the linker, or with the `editbin` Microsoft utility. You can use the `opp_stacktool` program (which relies on another Microsoft utility called `dumpbin`) to display reserved stack size for executables.

You need a low reserved stack size because the Win32 Fiber API which is the mechanism underlying `activity()` uses this number as coroutine stack size, and with 1MB being the default, it is easy to run out of the 2GB possible address space (2GB/1MB=2048).

A more detailed explanation follows. Each fiber has its own stack, by default 1MB (this is the ``reserved'' stack

space -- i.e. reserved in the address space, but not the full 1MB is actually ``committed'', i.e. has physical memory assigned to it). This means that a 2GB address space will run out after 2048 fibers, which is way too few. (In practice, you won't even be able to create this many fibers, because physical memory is also a limiting factor). Therefore, the 1MB reserved stack size (RSS) must be set to a smaller value: the coroutine stack size requested for the module, plus the `extra-stack-kb` amount for Cmdenv/Tkenv -- which makes about 16K with Cmdenv, and about 48K when using Tkenv. Unfortunately, the CreateFiber() Win32 API doesn't allow the RSS to be specified. The more advanced CreateFiberEx() API which accepts RSS as parameter is unfortunately only available from Windows XP.

The alternative is the stacksize parameter stored in the EXE header, which can be set via the STACKSIZE .def file parameter, via the /stack linker option, or on an existing executable using the editbin /stack utility. This parameter specifies a common RSS for the main program stack, fiber and thread stacks. 64K should be enough. This is the way simulation executable should be created: linked with the /stack:65536 option, or the /stack:65536 parameter applied using editbin later. For example, after applying the editbin /stacksize:65536 command to dyna.exe, I was able to successfully run the Dyna sample with 8000 Client modules on my Win2K PC with 256M RAM (that means about 12000 modules at runtime, including about 4000 dynamically created modules.)

``Segmentation fault''

On Unix, if you set the total stack size higher, you may get a segmentation fault during network setup (or during execution if you use dynamically created modules) for exceeding the operating system limit for maximum stack size. For example, in Linux 2.4.x, the default stack limit is 8192K (that is, 8MB). The `ulimit` shell command can be used to modify the resource limits, and you can raise the allowed maximum stack size up to 64M.

```
$ ulimit -s 65500
$ ulimit -s
65500
```

Further increase is only possible if you're root. Resource limits are inherited by child processes. The following sequence can be used under Linux to get a shell with 256M stack limit:

```
$ su root
Password:
# ulimit -s 262144
# su andras
$ ulimit -s
262144
```

If you do not want to go through the above process at each login, you can change the limit in the PAM configuration files. In Redhat Linux (maybe other systems too), add the following line to `/etc/pam.d/login`:

```
session    required    /lib/security/pam_limits.so
```

and the following line to `/etc/security/limits.conf`:

```
*    hard    stack    65536
```

A more drastic solution is to recompile the kernel with a larger stack limit. Edit `/usr/src/linux/include/linux/sched.h` and increase `_STK_LIM` from `(8*1024*1024)` to `(64*1024*1024)`.

Finally, it you're tight with memory, you can switch to Cmdenv. Tkenv increases the stack size of each module by about 32K so that user interface code that is called from a simple module's context can be safely executed. Cmdenv does not need that much extra stack.

Eventually...

Once you get to the point where you have to adjust the total stack size to get your program running, you should probably consider transforming (some of) your `activity()` simple modules to `handleMessage()`. `activity()` does not scale well for large simulations.

9.6.3 Memory leaks and crashes

The most common problems in C++ are associated with memory allocation (usage of `new` and `delete`):

- *memory leaks,* that is, forgetting to delete objects or memory blocks no longer used;
- *crashes,* usually due to referring to an already deleted object or memory block, or trying to delete one for a second time;
- *heap corruption* (eventually leading to crash) due to overrunning allocated blocks, i.e. writing past the end of an allocated array.

By far the most common ways leaking memory in simulation programs is by not deleting messages (`cMessage` objects or subclasses). Both Tkenv and Cmdenv are able to display the number of messages currently in the simulation, see e.g. section [9.2.4]. If you find that the number of messages is steadily increasing, you need to find where the message objects are. You can do so by selecting *Inspect|From list of all objects...* from the Tkenv menu, and reviewing the list in the dialog that pops up. (If the model is large, it may take a while for the dialog to appear.)

If the number of messages is stable, it is still possible you're leaking other `cOwnedObject`-based objects. You can also find them using Tkenv's *Inspect|From list of all objects...* function.

If you're leaking non-`cOwnedObject`-based objects or just memory blocks (`structs`, `int/double/struct` arrays, etc, allocated by `new`), you cannot find them via Tkenv. You'll probably need a specialized memory debugging tool like the ones described below.

Memory debugging tools

If you suspect that you may have memory allocation problems (crashes associated with double-deletion or accessing already deleted block, or memory leaks), you can use specialized tools to track them down.

By far the most efficient, most robust and most versatile tool is *Valgrind*, originally developed for debugging KDE.

Other memory debuggers are *NJAMD*, *MemProf*, *MPatrol*, *dmalloc* and *ElectricFence*. Most of the above tools support tracking down memory leaks as well as detecting double deletion, writing past the end of an allocated block, etc.

A proven commercial tool *Rational Purify*. It has a good reputation and proved its usefulness many times.

9.6.4 Simulation executes slowly

Check the following if you think your simulation is running too slow.

- Turn on express mode with the `cmdenv-express-mode=true` configuration option.
- Be sure that event logging is turned off (`record-eventlog=false` configuration option).
- Turn of vector file recording if you do not absolutely need it (`**.vector-recording=false`).
- If you are running under Tkenv disable animation features, close inspectors, hide the timeline, hide object tree, turn off log filtering.
- Compile your code as release instead of debug (in some cases this can give you 5x speedup)

What can you do if the simulation executes much slower than you expect? The best advice that can be given here is that you should **use a good profiler** to find out how much time is spent in each part of the program. Do not make the mistake of omitting this step, thinking that you know "which part is slow"! Even for experienced programmers, profiling session is all too often full of surprises. It often turns out that lots of CPU time is spent in completely

innocent-looking statements, while the big and complex algorithm doesn't take nearly as much time as expected. *Don't assume anything -- profile before you optimize!*

A great profiler on Linux is the *Valgrind*-based *callgrind*, and its visualizer *KCachegrind*. Unfortunately it won't be ported to Windows anytime soon. On Windows, you're out of luck -- commercial products may help, or, port your simulation to Linux. The latter goes usually much smoother than one would expect.

---

10 Network Graphics And Animation

10.1 Display strings

Display strings specify the arrangement and appearance of modules in graphical user interfaces (currently only Tkenv): they control how the objects (compound modules, their submodules and connections) are displayed. Display strings occur in NED description's `@display` property.

Display strings can be used in the following contexts:

- *submodules* -- display string 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 string can specify background color, border color, border thickness, background image, scaling, grid, unit of measurement, etc.
- *connections* -- display string can specify positioning, color, line thickness, line style, text and tooltip
- *messages* -- display string can specify icon, icon color, etc.

10.1.1 Display string syntax

The display string syntax is 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:

```
@display("p=100,100;b=,,rect,blue")
```

10.1.2 Display string placement

The following NED sample shows where to place display strings in the code:

```
simple Queue
{
    parameters:
        @display("i=block/queue");
    ...
}

network SimpleQueue
{
    parameters:
        @display("bgi=maps/europe");
    submodules:
        sink: Sink {
```

```
            @display("p=273,101");
        }
        ...
    connections:
        source.out --> { @display("ls=red,3"); } --> queue.in++;
}
```

10.1.3 Display string inheritance

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 `@display` properties are merged 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:

- If a tag is present in the base display string, but not in the current one the whole tag (with all arguments) is added to the current display string. (e.g. base: `"i=icon,red"` current: `"p=2,4"` result: `"p=2,4;i=icon,red"`)
- If a tag is present both in the base and in the current display string only tag arguments present in the base, but not in the current display string will be copied. (e.g. base: `"b=40,20"` current: `"b=,,oval"` result: `"b=40,20,oval"`)
- If the current display string contains a tag argument with value "-" (hyphen) that argument is treated as empty and will not be inherited from other display strings. Requesting the value of this argument will return its the default value.
- If neither the base display string nor the current one has value for a tag a suitable default value will be returned and used.

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"
        }
        ...
}
```

10.1.4 Display string tags used in submodule context

The following tags define how a module appears on the Tkenv user interface if it is used as a submodule:

- `b` -- shapes, colors
- `i` -- icon
- `is` -- icon size

`i2` -- alternate (status) icon placed at the upper right corner of the main icon

- `p` -- positioning and layouting
- `r` -- range indicator
- `q` -- queue information text
- `t` -- text
- `tt` -- tooltip

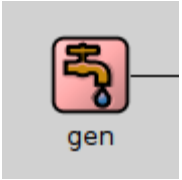
Icons

By default modules are represented by simple icons. Using images for the modules is possible with the `i` tag. See the `images` subfolder of your OMNeT++ installation for possible icons. The stock images installed with OMNeT++ have several size variants. Most of them have very small (vs), small (s), large (l) and very large (vl) variants. You can specify which variant you want to use with the `is` tag.

```
@display("i=block/source;is=l"); // a large source icon from the block icons group
```

Sometimes you want to have similar icons for modules, but would like to make them look a little different to create groups or to reflect status information about the module. You can easily change the color of an already existing image. The following example colorizes the `block/source` icon, 20% red

```
@display("i=block/source,red,20")
```



If you want to show state information about your module, you can use the `i2` tag to add a small status icon to your main icon. This icon is displayed in the upper right corner of your main icon. In most cases the `i2` tag is specified at runtime using the `setDisplayString()` method, so the icon can be changed dynamically based on the module's internal state.

```
@display("i=block/queue;i2=status/busy")
```

Shapes

If you want to have simple, but resizable representation for your module, you can use the `b` tag to create geometric shapes. Currently `oval` and `rectangle` is supported:

```
// an oval shape with 70x30 size, red background, black 4 pixel border
@display("b=70,30,oval,red,black,4")
```

Positioning, coordinates

To define the position of a module inside an other one use the `p` tag. If you do not specify a `p` tag for your module, the parent module will automatically choose a position based on a layouting algorithm. The following example will place the module at the given position:

```
@display("p=50,79");
```

> **NOTE**
> The coordinates specified in the `p`, `b` or `r` tags are not necessarily integers and measured in pixels. You can use the parent module's `bgs=`*`pix2unitratio,unit`* tag, to set the scaling parameter and the unit of measurement for your module. You can specify the ratio between 1 pixel and 1 unit with the `bgs` tag.

The `p` tag allows the automatic arrangement of module vectors. They can be arranged in a row, a column, a matrix or a ring or you may specify their positions later at runtime using the `setDisplayString()` method. The rest of the arguments in the `p` tag depends on the layout type:

- `row -- p=100,100,r,`*`deltaX`* (A row of modules with *deltaX* units between the modules)
- `column -- p=100,100,c,`*`deltaY`* (A column of modules with *deltaX* units between the modules)
- `matrix -- p=100,100,m,`*`noOfCols`*`,`*`deltaX`*`,`*`deltaY`* (A matrix with *noOfCols* columns. *deltaX* and *deltaY* units between rows and columns)
- `ring -- p=100,100,ri,`*`rx`*`,`*`ry`* (A ring (oval) with *rx* and *ry* as the horizontal and vertical radius.)
- `exact (default) -- p=100,100,x,`*`deltaX`*`,`*`deltaY`* (Place each module at *(100+deltaX, 100+deltaY)*. The coordinates are usually set at runtime.)

A matrix layout for a module vector:

```
@display("p=,,m,4,50,50");
```

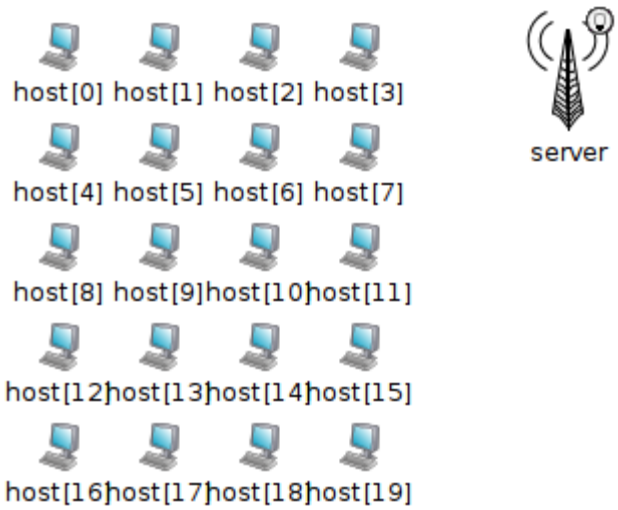


Figure: Matrix arrangement using the p tag

Wireless range

In wireless simulations it is very useful to show some kind of range around your module. This can be an interference range, transmission range etc. The following example will place the module at a given position, and draw a circle with a 90-unit radius around it as a range indicator:

```
submodules:
    ap: AccessPoint {
        @display("p=50,79;r=90");
    }
```

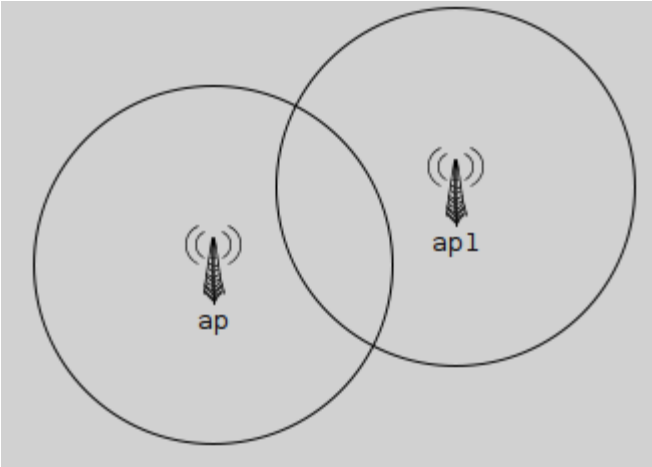
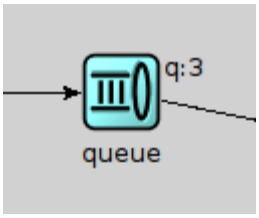


Figure: Range indicator using the r tag

Additional decorations

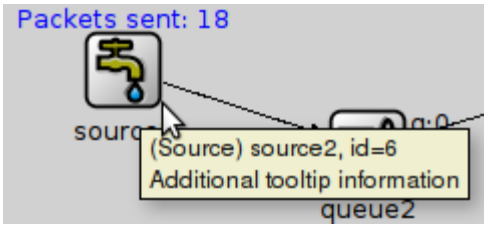
Sometimes you want to annotate your modules with additional information to make your model more transparent. One special case is when you want to show the length of a queue (`cQueue`) embedded somewhere in a module. In the following example the Server simple module contains a `cQueue` object, which was named by the `queue.setName("procqueue")` method. If we specify `q=procqueue` in the display string, Tkenv will descend into the module (several levels deep if needed) and look for a queue object named "procqueue". It will display the length of the queue object along the module.

```
@display("q=procqueue");
```



You can add a text description to any module using the `t` (displayed along the module) or `tt` tag (displayed as a tooltip). The following example displays a short text along with the module and adds a tooltip text too that can be seen by hovering over them module with the mouse.

```
@display("t=Packets sent: 18;tt=Additional tooltip information");
```



NOTE
Generally it makes no sense to assign static texts in the NED file. Usually the `t` and `tt` tags are used at runtime with the `setDisplayString()` method.

To see detailed descripton of the display string tags check Appendix [22] about display string tags.

10.1.5 Display string tags used in module background context

The following tags describe what a module looks like if opened in Tkenv. They mostly deal with the module background.

- `bgi` -- background image
- `bgtt` -- tooltip above the background
- `bgg` -- background grid
- `bgl` -- control child layouting
- `bgb` -- background size, color, border
- `bgs` -- scaling of background coordinates
- `bgp` -- background coordinate offset

In some cases it is required that the module's physical position should be modeled also in your model. In this case you would set the submodule display strings with the `setDisplayString()` method at run time, but it would be also nice to customize the area in what the modules are moving. It is possible to manipulate what a module's background looks like or whether we can use measurement units instead of pixels to set the position of the modules. The following example demonstrates the use of module background tags. The coordinates are given in km (SI unit). The `bgs=`*`pixelsperunit`*`,`*`unit`* specifies pixel/unit ratio, i.e. 1km is 0.075 pixel on the screen. The whole area is 6000x4500km (`bgb=`) and the map of Europe is used as a background and stretched to fill the module background. A light grey grid is drawn with a 1000km distance between major ticks, and 2 minor ticks per major tick (`bgg=tickdistance,minorpermajorticks,color`). See Figure below.

```
network EuropePlayground
{
    @display("bgb=6000,4500;bgi=maps/europe,s;bgg=1000,2,grey95;bgs=0.075,km");
```

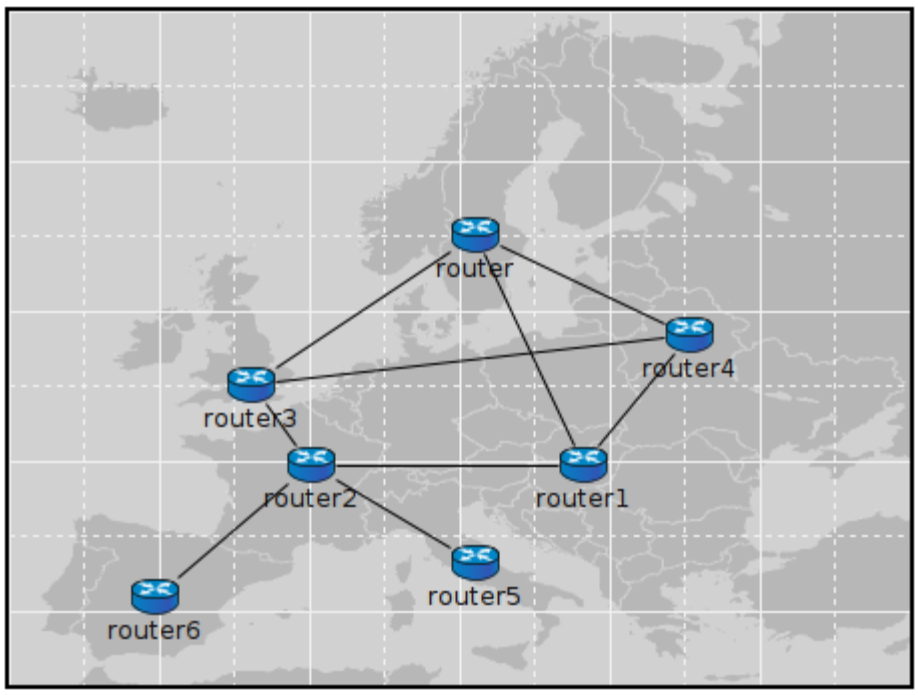


Figure: Background grid, scaling and image

After specifying the above `bgs` tag, all your submodule coordinates will be treated as if they were specified in km.

To see detailed descripton of the display string tags check Appendix [22] about display string tags.

10.1.6 Connection display strings

Connections may also have display strings. Connections inherit the display string property of their channel in the

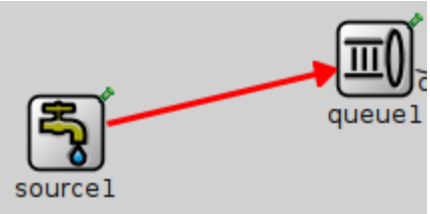
same way submodules inherit their display string from their types. If no channel is specified for a connection, `IdealChannel` will be used implicitly by the simulation kernel.

Connections support the following tags:

- `ls` -- line shape, and colors
- `t` -- text
- `tt` -- tooltip
- `m` -- used for hinting Tkenv how the connections should route to/from the module

Example of a thick, red connection:

```
source1.out --> { @display("ls=red,3"); } --> queue1.in++;
```



NOTE
If you want to hide a connection, specify width=0 for the connection. i.e. use `ls=,0`.

To see detailed descripton of these tags check Appendix [22] about display string tags.

10.1.7 Message display strings

Message display string affects 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 Tkenv Options dialog, and message kind dependent coloring can also be turned off there.

Specifying message display strings

Message objects do not store a display string by default, but you can redefine the `cMessage`'s `getDisplayString()` method and make it return one.

Example of using an icon to represent a message:

```
const char *CustomPacket::getDisplayString() const
{
    return "i=msg/packet;is=vs";
}
```

Or better: If you add the field `displayString` to your message definition (.msg file) the message compiler will automatically generate the `set/getDisplayString` methods for you:

```
message Job
{
    string displayString = "i=msg/package_s,kind";
...
```

Message display string tags

The following tags can be used in message display strings:

- b -- shapes, colors
- i -- icon
- is -- icon size

Using a small red box icon to represent the messages:

```
@display("i=msd/box,red;is=s");
```

Messages will be represented by a 15x15 rectangle with white background. Their border color will depend on the messageKind property of the message.

```
@display("b=15,15,rect,white,kind,5");
```

NOTE
In message display strings you may use the word kind as a special color. This virtual color depends on the messageKind field in the message.

10.2 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");
}
```

10.3 Colors

10.3.1 Color names

Any valid Tk color specification is accepted: English color names (blue, lightgrey, wheat) or *#rgb*, *#rrggbb* format (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.

You 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 to a color depending on the message kind. (See the getKind() method of cMessage.)

10.3.2 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 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.

10.4 The icons

10.4.1 The image path

In the current OMNeT++ version, module icons are PNG or GIF files. The icons shipped with OMNeT++ are in the `images/` subdirectory. Both the graphical NED editor and Tkenv need the exact location of this directory 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 Tkenv with the value "*omnetpp-dir*`/images; ./images;./bitmaps`" -- which will work fine as long as you don't move the directory, and you'll also be able to load more icons from the `images/` subdirectory of the current directory. As people usually 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 you're using the bash shell, adding a line

```
export OMNETPP_IMAGE_PATH="/home/you/images;./images"
```

to `~/.bashrc` or `~/.bash_profile` will do; on Windows, environment variables can be set via the *My Computer --> Properties* dialog.

You can extend the image path from `omnetpp.ini` with the `tkenv-image-path` option, which gets prepended to the environment variable's value.

```
[General]
tkenv-image-path = "/home/you/model-framework/images;/home/you/extra-images"
```

10.4.2 Categorized icons

Since OMNeT++ 3.0, icons are organized into several categories, represented by folders. These categories include:

- block/ - icons for subcomponents (queues, protocols, etc).
- device/ - network devices: servers, hosts, routers, etc.
- abstract/ - symbolic icons for various devices
- misc/ - node, subnet, cloud, building, town, city, etc.
- msg/ - icons that can be used for messages

Old (pre-3.0) icons are in the `old/` folder.

Tkenv and the IDE now load icons from subdirectories of all directories of the image path, and these icons can be referenced from display strings by naming the subdirectory (subdirectories) as well: `"subdir/icon"`,

`"subdir/subdir2/icon"`, etc.

For compatibility, if the display string contains a icon without a category (i.e. subdirectory) name, OMNeT++ tries it as "old/icon" as well.

10.4.3 Icon size

Icons come in various sizes: normal, large, small, very small. 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 whenever possible).

10.5 Layouting

OMNeT++ implements an automatic layouting feature, using a variation of the SpringEmbedder algorithm. Modules which have not been assigned explicit positions via the `"p="` tag will be automatically placed by the algorithm.

SpringEmbedder is a graph layouting algorithm based on a physical model. Graph nodes (modules) repent each other like electric charges of the same sign, and connections are sort of springs which try to contract and pull the nodes they're attached to. 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 get better results.

The algorithm doesn't move any module which has fixed coordinates. Predefined row, matrix, ring or other arrangements (defined via the 3rd and further args of the `"p="` tag) will be preserved -- you can think about them as if those modules were attached to a wooden framework so that they can only move as one unit.

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's a module with manually specified position, we don't want to move that one). To prevent rescaling, you can specify a sufficiently large bounding box in the background display string, e.g. `"b=2000,3000"`.
- Size is ignored by the present layouter, so longish modules (such as an Ethernet segment) may produce funny results.
- The algorithm is prone to produce erratic results, especially when the number of submodules is small, or when using predefined (matrix, row, ring, etc) layouts. The "Re-layout" toobar button can then be very useful. Larger networks usually produce satisfactory results.
- The algorithm is starting from random positions. To get the best results you may experiment with different seeds by specifying them using the `bgl=`*`seed`* display string tag.

10.6 Enhancing animation

10.6.1 Changing display strings at runtime

Often it is 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 you manipulate the string.

To get a pointer to the `cDisplayString` object, you can call the components's `getDisplayString()` method:

```
// Setting a module's position, icon and status icon:
cDisplayString *dispStr = getDisplayString();
dispStr->parse("p=40,20;i=device/cellphone;i2=status/disconnect");
```

NOTE
The connection display string is stored in the channel object, but it can also be accessed via the source gate of the connection.

```
// Setting an outgoing connection's color to red:
cDisplayString *gateDispStr = gate("out")->getDisplayString();
dispStr->parse("ls=red");
```

NOTE
In OMNeT++ 3.x, to manipulate the appearance of a compound module you had to use the `backgroundDisplayString()` method. This method is no longer supported in OMNeT++ 4.0, because there is no separate background display string. Use the `getDisplayString()` method instead with the background specific tags, i.e. those starting with `bg`.

```
// Setting module background and grid with background display string tags:
cDisplayString *parentDispStr = getParentModule()->getDisplayString();
parentDispStr->parse("bgi=maps/europe;bgg=100,2");
```

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.

An example:

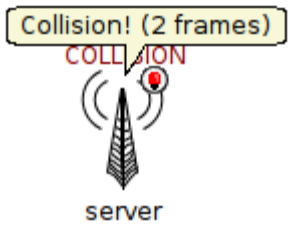
```
dispStr->parse("a=1,2;p=alpha,,3");
dispStr->insertTag("x");
dispStr->setTagArg("x",0,"joe");
dispStr->setTagArg("x",2,"jim");
dispStr->setTagArg("p",0,"beta");
ev << dispStr->str();  // result: "x=joe,,jim;a=1,2;p=beta,,3"
```

10.6.2 Bubbles

Modules can let the user know about important events (such as a node going down or coming up) by displaying a bubble with a short message ("Going down", "Coming up", etc.) This is done by the `bubble()` method of `cComponent`. The method takes the string to be displayed as a `const char *` pointer.

An example:

```
bubble("Collision! (2 frames)");
```



If the module contains a lot of code that modifies the display string or displays bubbles, it is recommended to make these calls conditional on `ev.isGUI()`. The `ev.isGUI()` call returns *false* when the simulation is run under Cmdenv, so one can make the code skip potentially expensive display string manipulation.

Better:

```
if (ev.isGUI())
    bubble("Going down!");
```

---

11 Analyzing Simulation Results

11.1 Result files

11.1.1 Results

vector results, scalar results

unique run id

attributes, e.g. experiment-measurement-replication

The format of result files is documented in detail in [24].

11.1.2 Output vectors

Output vectors are time series data: values with timestamps. You can use output vectors to record end-to-end delays or round trip times of packets, queue lengths, queueing times, link utilization, the number of dropped packets, etc. -- anything that is useful to get a full picture of what happened in the model during the simulation run.

Output vectors are recorded from simple modules, by `cOutVector` objects (see section [6.8.1]). Since output vectors usually record a large amount of data, in `omnetpp.ini` you can disable vectors or specify a simulation time interval for recording (see section [8.6.2]).

All `cOutVector` objects write to the same, common file. The following sections describe the format of the file, and how to process it.

11.1.3 Format of output vector files

An output vector file contains several series of data produced during simulation. The file is textual, and it looks like this:

```
# net-1.vec
```

```
vector 1   "subnet[4].term[12]"  "response time"  TV
1  12.895  2355.66666666
1  14.126  4577.66664666
vector 2   "subnet[4].srvr"  "queue length"  TV
2  16.960  2.00000000000.63663666
1  23.086  2355.66666666
2  24.026  8.00000000000.44766536
```

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.

vector data clustering for efficiency

indexed access

how to configure the IndexVectorFileManager (?)

11.1.4 Scalar results

Scalar results are recorded with `recordScalar()` calls, usually from the `finish()` methods of modules.

Format of scalar files

The corresponding output scalar file (by default, `omnetpp.sca`) will look like this:

```
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 `record()` call generates one "scalar" line in the file. (If you record statistics objects (`cStatictic` subclasses such as `cStdDev`) via their `record()` methods, they'll generate several lines: mean, standard deviation, etc.) In addition, several simulation runs can record their results into a single file -- this facilitates comparing them, creating x-y plots (*offered load vs throughput*-type diagrams), etc.

11.2 The Analysis Tool in the Simulation IDE

The Simulation IDE provides an Analysis Tool for analysis and visualization of simulation results. The Analysis Tool lets you load several result files at once, and presents their contents somewhat like a database. You can browse the results, select the particular data you are interested in (scalars, vectors, histograms), apply processing steps, and create various charts or plots from them. Data selection, processing and charting steps can be freely combined, resulting in a high degree of freedom. These steps are grouped into and stored as "recipes", which get automatically re-applied when new result files are added or existing files are replaced. This automation spares the user lots of repetitive manual work, without resorting to scripting.

The Analysis Tool is covered in detail in the User Guide.

11.3 Scave Tool

Much of the IDE Analysis Tool's functionality is available on the command line as well, via the `scavetool` program. `scavetool` is suitable for filtering and basic processing of result files, and exporting the result in various formats digestible for other tools. `scavetool` has no graphics capabilities, but it can be used to produce files that

can be directly plotted with other tools like gnuplot (see [11.4.6]).

When `scavetool` is invoked without arguments, it prints usage information:

```
scavetool <command> [options] <file>...
```

11.3.1 Filter command

The result files can be filtered and processed by this command and the result can be written into files as vector files, CSV files, Matlab or Octave files.

The filter can be specified by the \itshape -p <filter> option. The filter is one or more \itshape <fieldname>(<pattern>) expression connected with AND, OR and NOT operators. The possible field names are:

- **file**: full path of the result file
- **run**: run identifier
- **module**: module name
- **name**: vector name
- **attr:<runAttribute>**: value of an attribute of the run (e.g. experiment)
- **param:<moduleParameter>**: value of the parameter in the run

Processing operations can be applied to vectors by the \itshape -a <function>(<parameterlist>) option. You can list the available functions and their parameters by the \itshape info command.

The name and format of the output file can be given by the \itshape -O <file> and \itshape -F <formatname> options, where the formatname is one of:

- **vec**: vector file (default)
- **csv**: CSV file
- **octave**: Octave text file
- **matlab**: Matlab script file

Examples:

```
scavetool filter -p "queuing time" -a winavg(10) -O out.vec in.vec
```

Writes the window averaged queuing times stored in in.vec into out.vec.

```
scavetool filter -p "module(**.sink) AND
                    (\"queueing time\" OR \"transmission time\")"
                 -O out.csv -F csv in.vec
```

Writes the queing and transmission times of sink modules into CSV files. It generates a separate files for each vector named *out-1.csv*, *out-2.csv*... The generated file contains a header and two columns:

```
time,"Queue.sink.queueing time"
2.231807576851,0
7.843802235089,0
15.797137536721,3.59449
21.730758362277,6.30398
[...]
```

11.3.2 Index command

If the index file was deleted or the vector file was modified, you need to rebuild the index file before running the filter command.

Normally the vector data is written in blocks into the vector file. However if the vector file was generated by an older version of the `cOutputVectorManager` it might not be so. In this case you have to specify the -r option to rearrange the records of the vector file, otherwise the index file would be too big and the indexing inefficient.

11.3.3 Summary command

The summary command reports the list of statistics names, module names, run ids, configuration names in the given files to the standard output.

11.4 Alternative statistical analysis and plotting tools

There are several programs and packages that can also be used to analyse result files, and create various plots and charts from them.

11.4.1 Spreadsheet programs

One straightforward solution is to use spreadsheets such as OpenOffice Calc, Microsoft Excel, Gnumeric or KSpread. Data can be imported from csv or other formats, exported with `scavetool` (see [11.3]).

Spreadsheets have good charting and statistical features. A useful functionality spreadsheets offer for analysing scalar files is *PivotTable* (Excel) or *DataPilot* (OpenOffice). The drawback of using spreadsheets is limited automation, leading to tedious and repetitive tasks; also, the number of rows is usually limited to about 32,000..64,000, that can be painful when working with large vector files.

11.4.2 GNU R

R is a free software environment for statistical computing and graphics. R has an excellent programming language and powerful plotting capabilities, and it is supported on all major operating systems and platforms.

R is widely used for statistical software development and data analysis. The program uses a command line interface, though several graphical user interfaces are available.

Several OMNeT++-related packages such as SimProcTC and Syntony already use R for data analysis and plotting. In the future, OMNeT++ is going to be extended with features that further facilitate using it with R.

11.4.3 MATLAB or Octave

MATLAB is a commercial numerical computing environment and programming language. MATLAB allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages.

Octave is an open-source Matlab-like package, available on nearly all platforms. Currently Octave relies on Gnuplot for plotting, and has more limited graphics capabilities than GNU R or MATLAB.

11.4.4 NumPy and MatPlotLib

MatPlotlib is a plotting library for the Python programming language and its NumPy numerical mathematics extension. It provides a "pylab" API designed to closely resemble that of MATLAB, thereby making it easy to learn for experienced MATLAB users. Matplotlib is distributed under a BSD-style license.

11.4.5 ROOT

ROOT is a powerful object-oriented data analysis framework, with strong support for plotting and graphics in general. ROOT was developed at CERN, and is distributed under a BSD-like license.

ROOT is based on *CINT*, a ``C/C++ interpreter'' aimed at processing C/C++ scripts. It is probably harder to get started using ROOT than with either Gnuplot or Grace, but if you are serious about analysing simulation results, you will find that ROOT provides power and flexibility that would be unattainable the other two programs.

Curt Brune's page at Stanford (http://www.slac.stanford.edu/~curt/omnet++/) shows examples what you can achieve using ROOT with OMNeT++.

11.4.6 Gnuplot

Gnuplot is a very popular command-line program that can generate two- and three-dimensional plots of functions and data. The program runs on all major platforms, and it is well supported.

Gnuplot has an interactive command interface. For example, if you have the data files `foo.csv` and `bar.csv` that contain two values per line (*x y*; such files can be exported with `scavetool` from vector files), you can plot them in the same graph by typing:

```
plot "foo.csv" with lines, "bar.csv" with lines
```

To adjust the *y* range, you would type:

```
set yrange [0:1.2]
replot
```

Several commands are available to adjust ranges, plotting style, labels, scaling etc. On Windows, you can copy the resulting graph to the clipboard from the Gnuplot window's system menu, then insert it into the application you are working with.

11.4.7 Grace

Grace (also known as *xmgrace*, a successor of *ACE/gr* or *Xmgr*) is a powerful GPL data visualization program with a menu-and-dialog graphical user interface for X and Motif. It has also been ported to Windows.

Grace can export graphics in various raster and vector formats, and has many useful features like built-in statistics and analysis functions (e.g. correlation, histogram), fitting, splines, etc., and it also has a built-in programming language.

---

12 Eventlog

12.1 Introduction

The eventlog feature and the related tools are completely new in OMNeT++ 4.0. They aim to help understanding complex simulation models and to help correctly implementing the desired component behaviors. Using these tools you will be able to easily examine every minute detail of the simulation back and forth in terms of simulation time or step-by-step focusing on the behavior instead of the statistical results of your model.

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 OMNeT++ 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 show you only the relevant parts focusing on 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), because it contains a lot of information about the run, especially when message detail recording is turned on.

12.2 Configuration

To record an eventlog file during the simulation, simply insert the following line into the ini file.

```
record-eventlog = true
```

> **NOTE**
> Since writing an eventlog file might significantly decrease overall simulation performance therefore eventlog recording is turned off by default.

12.2.1 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
```

12.2.2 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 following ini file configuration entry instructs the kernel to record events with simulation time less than 10.2, between 22.2 and 100 and greater than 233.3.

```
eventlog-recording-intervals = ..10.2, 22.2..100, 233.3..
```

12.2.3 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 following ini file configuration entry instructs the kernel to record only the events that occurred in any of the routers having index between 10 and 20 while turns off recording for all other modules. This configuration key makes sense only for simple modules.

```
**.router[10..20].**.module-eventlog-recording = true
**.module-eventlog-recording = false
```

12.2.4 Recording Message Data

Since recording message data dramatically increases the size of the eventlog file and additionally slows down the simulation therefore it is turned off by default even if writing the eventlog is enabled. To turn on message data recording enter a value for the following configuration key in the ini file:

```
eventlog-message-detail-pattern
```

An example configuration for an IEEE 80211 model that records the field encapsulationMsg and all other fields which name ends with Address from messages which class name ends with 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:encapsulatedMsg|IPDatagram:encapsulatedMsg,*Address|TCPSegment:*Port
```

NOTE
Take care about long running simulations because they might output eventlog files of several Gbytes and thus fail due to insufficient disk space.

12.3 Eventlog Tool

The Eventlog Tool is a command line tool to process eventlog files. Invoking it without parameters will display usage information such as available commands and options. The following are the most useful commands for users.

12.3.1 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 OMNeT++ 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.

12.3.2 Echo

Since the eventlog file format is text based and users are encouraged to implement their own filters therefore there needs to be a way 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 OMNeT++ IDE.

NOTE
Custom filter tools should filter out whole events only otherwise the consequences are undefined.

---

13 Documenting NED and Messages

13.1 Overview

OMNeT++ 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, unassigned 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 you also generate the C++ documentation 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. Of course, one still has to write documentation comments in the code.

In the 4.0 version, the documentation tool is part of the Eclipse-based simulation IDE.

13.2 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.

> [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 OMNeT++ there's 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.]

Example:

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//
simple Gen
{
    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
}
```

You can also place comments above parameters and gates. This is useful if they need long explanations. Example:

```
//
// Deletes packets and optionally keeps statistics.
//
simple Sink
{
    parameters:
        // You can turn statistics generation on and off. This is
        // a very long comment because it has to be described what
        // statistics are collected (or not).
        bool collectStatistics = default(true);
    gates:
        input in;
}
```

13.2.1 Private comments

If you want a comment line *not* to appear in the documentation, begin it with `//#`. Those lines will be ignored by the documentation tool, and 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
}
```

13.2.2 More on comment placement

Comments should be written where nedtool 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's 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.

13.3 Text layout and formatting

13.3.1 Paragraphs and lists

If you write longer descriptions, you'll need text formatting capabilities. Text formatting works like in Javadoc or Doxygen -- you can break up the text into paragraphs and create bulleted/numbered lists without special commands, and use HTML for more fancy formatting.

Paragraphs are separated by empty lines, like in LaTeX or Doxygen. Lines beginning with `-' will be turned into bulleted lists, and lines beginning with `-#' into numbered lists.

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
```

```
//
```

13.3.2 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
//
```

13.3.3 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'').

If you insert links to external pages (web sites), its useful to add the `target="_blank"` attribute to ensure pages come up in a new browser window and not just in the current frame which looks awkward. (Alternatively, you 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>.
//
```

You 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>
// ...
//
```

You 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 (or you can turn them 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

13.3.4 Escaping HTML tags

Sometimes may need to off interpreting HTML tags (, <b>, etc.) as formatting instructions, and rather you want them to appear as literal , <b> texts in the documentation. You can achieve this via 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 tag (like this) 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* if you happen to have an IP module in the NED files.

13.4 Customizing and adding pages

13.4.1 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 generic title ``*OMNeT++ Model Documentation*''. You probably want to customize that, and at least change the title to the name of the documented simulation model.

You can supply your own version of the title page adding a `@titlepage` directive to a file-level comment (a comment that appears at the top of a NED file, but is separated from the first `import`, `channel`, `module`, etc. definition by at least one blank line). In theory you can place your title page definition into any NED or MSG file, but it is probably a good idea to create a separate `index.ned` file for it.

The lines you write after the `@titlepage` line up to the next `@page` line (see later) or the end of the comment will be used as the title page. You probably want to begin with a title because the documentation tool doesn't add one (it lets you have full control over the page contents). You can use the `<h1>..</h1>` HTML tag to define a title.

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.
//
```

13.4.2 Adding extra pages

You can add new pages to the documentation in a similar way as customizing the title page. The directive to be used is `@page`, and it can appear in any file-level comment (see above).

The syntax of the `@page` directive is the following:

```
// @page filename.html, Title of the Page
```

Please choose a file name that doesn't collide with the files generated by the documentation tool (such as `index.html`). The page title you supply will appear on 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. You don't need to add a title because the documentation tool automatically adds one.

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
// ...
//
```

You 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 you don't have to worry about specifying directories.

Example:

```
//
// @titlepage
// ...
// The structure of the model is described <a href="structure.html">here</a>.
//
```

13.4.3 Incorporating externally created pages

You may want to create pages outside the documentation tool (e.g. using a HTML editor) and include them in the documentation. This is possible, all you have to do is declare such pages with the `@externalpage` directive in any of the NED files, and they will be added to the page index. The pages can then be linked to from other pages using the HTML `<a href="...">...</a>` tag.

The `@externalpage` directive is similar in syntax to `@page`:

```
// @externalpage filename.html, Title of the Page
```

The documentation tool does not check if the page exists or not. It is your responsibility to copy it manually into the directory of the generated documentation, and to make sure the hyperlink works.

---

14 Parallel Distributed Simulation

14.1 Introduction to Parallel Discrete Event Simulation

OMNeT++ supports parallel execution of large simulations. The following paragraphs provide 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, thus they will maintain their own local simulation times. 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:

- **Conservative algorithms** prevents incausalities from happening. The Null Message Algorithm exploits knowledge of the time when LPs send messages to other LPs, and uses `null' messages to propagate this information to other LPs. If an LP knows it won't receive any messages from other LPs until *t+Δ t* simulation time, it may advance until *t+Δ t* without the need for external synchronization. Conservative simulation tends to converge to sequential simulation (slowed down by communication between LPs) if there's not enough parallelism in the model, or parallelism is not exploited by sending a sufficient number of `null' messages.

- **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 OMNeT++ 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.

14.2 Assessing available parallelism in a simulation model

OMNeT++ currently supports conservative synchronization via the classic Chandy-Misra-Bryant (or null message) algorithm [chandymisra79]. 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).
  [Notations: *ev:* events, *sec:* real seconds, *simsec:* simulated seconds]
  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 [ongfarrell2000]. Specialized hardware such as Quadrics Interconnect [quadrics] 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 OMNeT++ displays these values on the GUI while the simulation is running, see Figure below. Cmdenv can also be configured to display these values.

Figure: Performance bar in OMNeT++ 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 τ *P*:

$\lambda = (LE) / (\tau P)$

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 change that the simulation will perform well when run in parallel. With $\lambda < 1$, poor performance is guaranteed. For details see the paper [ParsimCrit03].

14.3 Parallel distributed simulation support in OMNeT++

14.3.1 Overview

This chapter presents the parallel simulation architecture of OMNeT++. 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 OMNeT++ 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, OMNeT++ 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.

OMNeT++ 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 [bagrodia00]. 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 OMNeT++, 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), OMNeT++ primarily uses MPI, the Message Passing Interface standard [mpiforum94]. 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 OMNeT++ 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 ones and select them in the configuration.

14.3.2 Parallel Simulation Example

We will use the Parallel CQN example simulation for demonstrating the PDES capabilities of OMNeT++. The model consists of *N* tandem queues where each tandem consists of a switch and *k* single-server queues with exponential

service times (Figure below). 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 OMNeT++ model for CQN wraps tandems into compound modules.

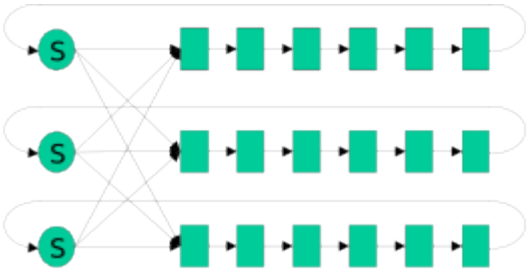


Figure: The Closed Queueing Network (CQN) model

To run the model in parallel, we assign tandems to different LPs (Figure below). Lookahead is provided by delays on the marked links.

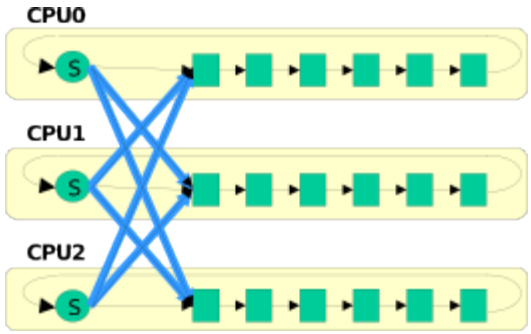


Figure: Partitioning the CQN model

To run the CQN model in parallel, we have to configure it for parallel execution. In OMNeT++, the configuration is in a text file called `omnetpp.ini`. 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.

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 [lammpi], 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 OMNeT++ utility can be used to start the processes. Alternatively, one can run the processes by hand (the -p flag tells OMNeT++ 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. (OMNeT++ provides the necessary configuration options.)

The graphical user interface of OMNeT++ can also be used (as evidenced by Figure below), independent of the selected communication mechanism. The GUI interface can be useful for educational or demonstration purposes. OMNeT++ displays debugging output about the Null Message Algorithm, EITs and EOTs can be inspected, etc.

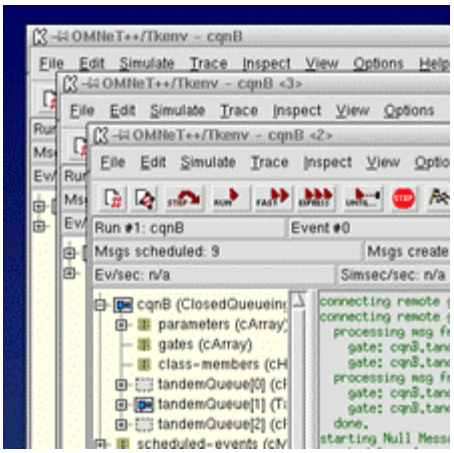


Figure: Screenshot of CQN running in three LPs

14.3.3 Placeholder modules, proxy gates

When setting up a model partitioned to several LPs, OMNeT++ 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 below).

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 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 below).

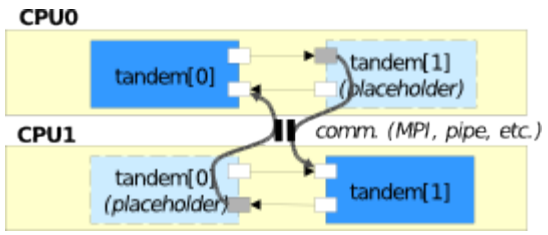


Figure: Placeholder modules and proxy gates

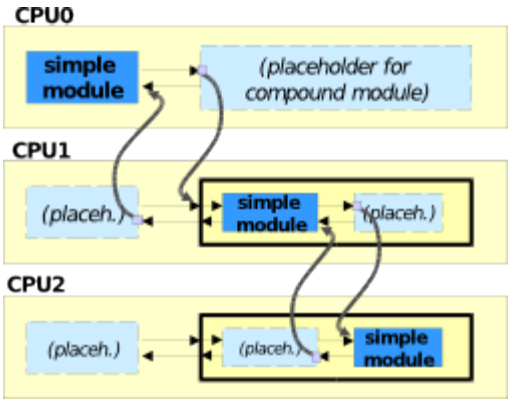


Figure: Instantiating compound modules

14.3.4 Configuration

Parallel simulation configuration is the `[General]` section of `omnetpp.ini`.

The parallel distributed simulation feature can be turned on with the **`parallel-simulation`** 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 [23].

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 do not overwrite each other's output files. (See section [9.5.3])

14.3.5 Design of PDES Support in OMNeT++

Design of PDES support in OMNeT++ follows a layered approach, with a modular and extensible architecture. The overall architecture is depicted in Figure below.

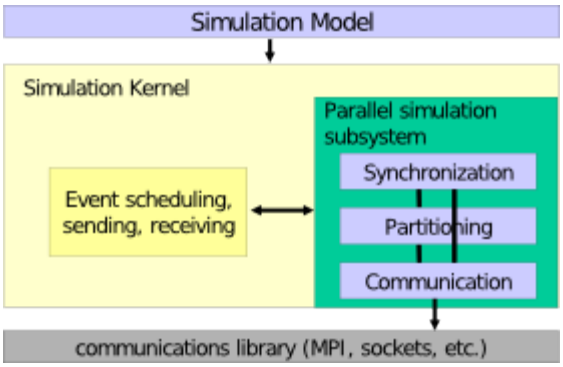


Figure: Architecture of OMNeT++ PDES implementation

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 [bagrodia00] 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 ones.

The Ideal Simulation Protocol (ISP; see [bagrodia00]) implementation consists in fact 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

> [Unfortunately, support for state saving/restoration needs to be individually and manually added to each class in the simulation, including user-programmed simple modules.]

.

We also expect that because of the modularity, extensibility and clean internal architecture of the parallel simulation subsystem, the OMNeT++ framework has the potential to become a preferred platform for PDES research.

---

15 Plug-in Extensions

15.1 Overview

OMNeT++ is an open system, and several details of its operation can be customized via plug-ins. To create a plug-in, you generally need to write a C++ class that implements a certain interface (i.e. subclasses from a C++ abstract class), and register it in OMNeT++. The plug-in class can be activated for a particular simulation with a corresponding configuration option.

The following plug-in interfaces are supported:

- `cRNG`. Interface for random number generators.
- `cScheduler`. The scheduler class. This plug-in interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation.
- `cConfigurationEx`. Configuration provider plug-in. This plug-in interface lets you replace `omnetpp.ini` with some other implementation, for example a database.
- `cOutputScalarManager`. It handles recording the scalar output data. The default output scalar manager is `cFileOutputScalarManager`, defined in the Envir library.
- `cOutputVectorManager`. It handles recording the output from `cOutVector` objects. The default output vector manager is `cIndexedFileOutputVectorManager`, defined in the Envir library.
- `cSnapshotManager`. It provides an output stream to which snapshots are written (see section [6.9.5]). The default snapshot manager is `cFileSnapshotManager`, defined in the Envir library.

The classes (`cRNG`, `cScheduler`, etc.) are documented in the API Reference.

To actually implement and select a plug-in for use:

□□□ Subclass the given interface class (e.g. for a custom RNG, `cRNG`) to create your own version.
□□□ Register the class by putting the `Register_Class(MyRNGClass)` line into the C++ source.
□□□ Compile and link your interface class into the OMNeT++ simulation executable. IMPORTANT: make sure the executable actually contains the code of your class! Over-optimizing linkers (esp. on Unix) tend to leave out code to which there seem to be no external reference.
□□□ Add an entry to `omnetpp.ini` to tell Envir use your class instead of the default one. For RNGs, this setting is `rng-class` in the `[General]` section.

15.2 Plug-in descriptions

15.2.1 Defining a new random number generator

The new RNG C++ class must implement the `cRNG` interface, and can be activated with the `rng-class` configuration option.

15.2.2 Defining a new scheduler

The scheduler plug-in interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation. The scheduler C++ class must implement the `cScheduler` interface, and can be activated with the `scheduler-class` configuration option.

To see examples of scheduler classes, check the `cRealTimeScheduler` class in the simulation kernel, and `cSocketRTScheduler` which is part of the *Sockets* sample simulation.

15.2.3 Defining a new configuration provider

Overview

The configuration provider plug-in lets you 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 a little. It assumes that the configuration data consists of several *named configurations*. Before every simulation run, one of the *named configurations* get activated, and from then on, all queries into the configuration operate on the *active named configuration* only.

It practice, you'll probably use the `SectionBasedConfiguration` class (in `src/envir`) or subclass from it, because it already implements a lot of functionality that otherwise you would 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 provides the data in raw form to it. The default implementation of `cConfigurationReader` is `InifileReader`.

The startup sequence

From the configuration plug-in's point of view, the startup sequence looks like the following (see `src/sim/bootenv.cc` in the source code):

- 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`.
- Shared libraries get loaded (see the `-l` command-line option, and the `load-libs` configuration option). This allows configuration classes to come from shared libraries.
- 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.
- The `load-libs` option from the new configuration object is processed.
- Then everything goes on as normally, using the new configuration object.

Providing a custom configuration class

To replace the configuration object with your custom implementation, you would write the class:

```
#include "cconfiguration.h"

class CustomConfiguration : public cConfigurationEx
{
   ...
};

Register_Class(CustomConfiguration);
```

and then activate it in the boot-time configuration:

```
[General]
configuration-class = CustomConfiguration
```

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, interpreted by `SectionBasedConfiguration`. Specify the following in your boot-time ini file:

```
[General]
configuration-class = SectionBasedConfiguration
sectionbasedconfig-configreader-class = <my new reader class>
```

The configuration reader class should look like this:

```
#include "cconfigreader.h"

class DatabaseConfigurationReader : public cConfigurationReader
{
   ...
};

Register_Class(DatabaseConfigurationReader);
```

15.2.4 Defining a new output scalar manager

cOutputScalarManager handles recording the scalar output data. The default output scalar manager is `cFileOutputScalarManager`, defined in the Envir library.

The new class can be activated with the `outputscalarmanager-class` configuration option.

15.2.5 Defining a new output vector manager

cOutputVectorManager handles recording the output from `cOutVector` objects. The default output vector manager is `cIndexedFileOutputVectorManager`, defined in the Envir library.

The new class can be activated with the `outputvectormanager-class` configuration option.

15.2.6 Defining a new snapshot manager

cSnapshotManager provides an output stream to which snapshots are written (see section [6.9.5]). The default

snapshot manager is `cFileSnapshotManager`, defined in the Envir library.

The new class can be activated with the `snapshotmanager-class` configuration option.

15.3 Accessing the configuration

config is the ini file plus --ccc=vvv options

15.3.1 Defining new configuration options

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, TYPE, DEFAULTVALUE, DESCRIPTION)
Register_PerObjectConfigOptionU(ID, NAME, UNIT, DEFAULTVALUE, DESCRIPTION)
```

Config options come in three flavours, 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)
- *Per-Object* options ...

The macro arguments are as follows:

- *ID* is a C++ identifier that will let you refer to the configuration option in *cConfiguration* member functions. (It is actually pointer to a `cConfigOption` object that the macro creates.)
- *NAME* is the name of the option (a string).
- *TYPE* is the data type of the 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 string is that relative filenames and paths get 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 `NULL` 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` macro is declared in the following way:

```
Register_GlobalConfigOption(CFGID_DEBUG_ON_ERRORS,
                            "debug-on-errors", CFG_BOOL, "false",
                            "When enabled, runtime errors will etc etc...");
```

NOTE
Registration is necessary because from the 4.0 version, OMNeT++ validates the configuration on startup, in order to be able to report invalid or mistyped option names and other errors.

15.3.2 Reading values from the configuration

The configuration is accessible via the `getConfig()` method of `cEnvir`. It returns a pointer to the configuration object (`cConfiguration`):

```
cConfiguration *config = ev.getConfig();
```

`cConfiguration` provides several methods for querying the configuration.

no sections: flattened view etc.

```
const char *getAsCustom(cConfigOption *entry, const char *fallbackValue=NULL);
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 = ev.getConfig()->getAsBool(CFGID_PARSIM_DEBUG);
```

15.4 Implementing a new user interface

It is possible to extend OMNeT++ with a new user interface. The new user interface will have fully equal rights to Cmdenv and Tkenv, 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 OMNeT++ with the `Register_OmnetApp()` macro. In practice, you'll 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 when embedding the simulation kernel into another application, then you should be reading section [16.2], 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 get 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 behaviour. One method that you'll 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.

---

16 Embedding the Simulation Kernel

16.1 Architecture

OMNeT++ has a modular architecture. The following diagram shows the high-level architecture of OMNeT++ simulations:

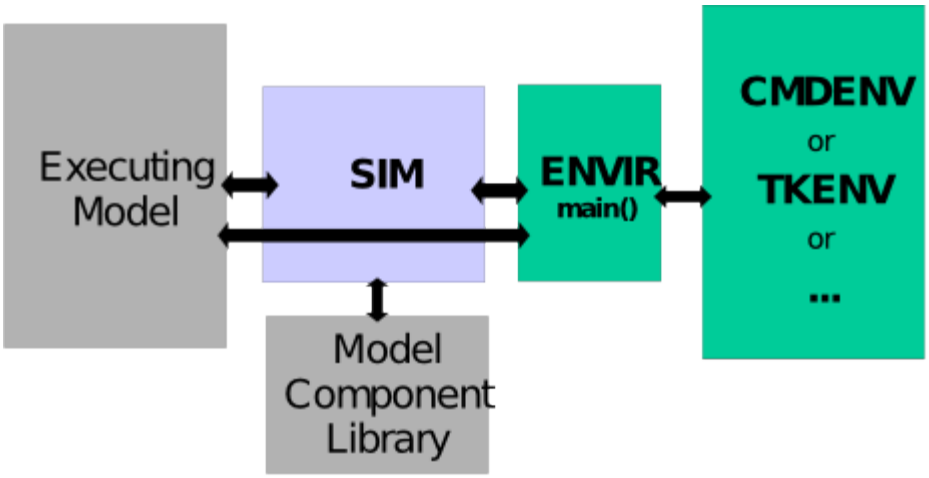


Figure: Architecture of OMNeT++ simulation programs

The rectangles in the picture represent components:

- **Sim** is the simulation kernel and class library. Sim exists as a library you link your simulation program with.
- **Envir** is another library which contains all code that is common to all user interfaces. `main()` is also in Envir. Envir provides services like ini file handling for specific user interface implementations. Envir presents itself towards Sim and the executing model via the `ev` facade object, hiding all other user interface internals. Some aspects of Envir can be customized via plugin interfaces. Embedding OMNeT++ into applications can be achieved by implementing a new user interface in addition to Cmdenv and Tkenv, or by replacing Envir with another implementation of `ev` (see sections [15.4] and [16.2].)
- **Cmdenv and Tkenv** are specific user interface implementations. A simulation is linked with Cmdenv, Tkenv, or both.
- The **Model Component Library** consists of simple module definitions and their C++ implementations, compound module types, channels, networks, message types and in general everything that belongs to models and has been linked into the simulation program. A simulation program is able to run any model that has all necessary components linked in.
- The **Executing Model** is the model that has been set up for simulation. It contains objects (modules, channels, etc.) that are all instances of components in the model component library.

The arrows in the figure show how components interact with each other:

- **Executing Model <==> Sim**. The simulation kernel manages the future events and invokes modules in the

executing model as events occur. The modules of the executing model are stored in the main object of Sim, `simulation` (of 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. It also refers to the component library when dynamic module creation is used. The machinery for registering and looking up components in the model component library is implemented as part of Sim.
- **Executing Model <==> Envir**. The `ev` object, logically part of Envir, is the facade of the user interface towards the executing model. The model uses `ev` to write debug logs (`ev<<`, `ev.printf()`).
- **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 (`ev`) that represents the environment (user interface) toward Sim -- no Envir internals are visible to Sim or the executing model. During simulation model setup, Envir supplies 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 <==> Tkenv/Cmdenv**. Tkenv and Cmdenv 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 by invoking its `run()` method. Sim's or the model's calls on the `ev` object are delegated to the user interface.

16.2 Embedding the OMNeT++ simulation kernel

This section discusses the issues of embedding the simulation kernel or a simulation model into a larger application. We assume that you don't just want to change one or two aspects of the simulator (like event scheduling or result recording) or create a new user interface a'la Cmdenv or Tkenv -- if so, see chapter [15].

For the following discussion, we assume that you write the embedding program from scratch, i.e. starting from a `main()` function.

16.2.1 The main() function

Here's an minimalistic program that initializes the simulation library, and runs two simulations. In later sections we'll go through details of the code and discuss how to elaborate it.

```
#include <omnetpp.h>

int main(int argc, char *argv[])
{
    // the following line MUST be at the top of main()
    cStaticFlag dummy;

    // initializations
    ExecuteOnStartup::executeAll();
    SimTime::setScaleExp(-12);

    // load NED files
    cSimulation::loadNedSourceFolder("./foodir");
    cSimulation::loadNedSourceFolder("./bardir");
    cSimulation::doneLoadingNedFiles();

    // run two simulations
    simulate("FooNetwork", 1000);
    simulate("BarNetwork", 2000);
```

```
    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. `ExecuteOnStartup::executeAll()` does 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 simulation time resolution to picoseconds; other values can be used as well, but it is mandatory to choose one.

> **NOTE**
> It is not allowed to change the simulation time exponent later, because as the exponent is a global variable, existing `simtime_t` instances would change their values.

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 goes on to run two simulations.

16.2.2 The simulate() function

A minimalistic version of the `simulate()` function is shown below. For conciseness, we omitted exception handling code (`try/catch` blocks) except for the event loop, but marked every line with ``E!'' where various problems with the simulation model can manifest as exceptions.

```
void simulate(const char *networkName, simtime_t limit)
{
    // look up network type
    cModuleType *networkType = cModuleType::find(networkName);
    if (networkType == NULL) {
        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->startRun(); //E!

    // run the simulation
    bool ok = true;
    try {
        while (sim->getSimTime() < limit) {
            cSimpleModule *mod = sim->selectNextModule(); //E!
            if (!mod)
                break;
            sim->doOneEvent(mod);  //E!
        }
        printf("Finished: time limit reached\n");
    }
    catch (cTerminationException& e) {
        printf("Finished: %s\n", e.what());
    }
    catch (std::exception& e) {
        ok = false;
        printf("ERROR: %s\n", e.what());
    }

    if (ok)
```

```
        simulation.callFinish();  //E!

    // finish the simulation and clean up the network
    sim->endRun();  //E!
    sim->deleteNetwork();  //E!

    cSimulation::setActiveSimulation(NULL);
    delete sim; // deletes env as well
}
```

The function accepts a network type name (which must be fully qualified, i.e. with package name), and a simulation time limit.

In the first few lines we look up the network name among the modules that have been loaded from NED files, and print an error message if not found.

Then we need to create and activate a simulation manager object (`cSimulation`). The simulation manager needs an environment object to take the configuration from, write simulation results to, and so on. The environment object (`CustomSimulationEnv` in the above code) has to be provided by the programmer; this is discussed in detail in a later section.

> **NOTE**
> Before the 4.0 version, `simulation` and `ev` were global variables; now they are macros that resolve to `*cSimulation::getActiveSimulation()` and `*cSimulation::getActiveSimulation()->getEnvir()`.

Then we set up the network 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 needed) and assigned. If there is an error (e.g. module type not found), an exception will be thrown. The exception object is some kind of `std::exception`, usually a `cRuntimeError`.

If network setup was successful, we call `sim->startRun()`, one side effect of which is that the `initialize()` methods of modules and channels get invoked. This step also results in an exception if something goes wrong in an `initialize()`.

The following lines actually run the simulation, by calling `sim->selectNextModule()` and `sim->doOneEvent()` in an event loop, until the simulation time limit is reached or some exception occurs. Exceptions subclassed from `cTerminationException` signal the normal termination of the simulation; other exceptions signal various errors.

If the simulation has completed successfully (`ok==true`), the code goes on to call the `finish()` methods of modules and channels. Then, regardless of whether there was an error, `sim->endRun()` has to be called, and the network is torn down with `sim->deleteNetwork()`.

Finally, we deallocate the simulation manager object, but the active simulation manager is not allowed to be deleted, so we first deactivate it with `setActiveSimulation(NULL)`.

16.2.3 Providing an environment object

The environment object needs to be subclassed from the `cEnvir` class, but since it has quite a number of pure virtual methods, it is easier to start 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 are progressing with the development.

One method that you surely want to redefine is `readParameter()`; that's how module parameters get their values. For debugging purposes, you may also redefine `sputn()` where module log messages are written. `cNullEnvir` only provides one random number generator, so 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's 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);
    }
};
```

16.2.4 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 NULL;}
        virtual const char *getValue() const {return NULL;}
        virtual const char *getBaseDirectory() const {return NULL;}
    };
    NullKeyValue nullKeyValue;

  protected:
    virtual const char *substituteVariables(const char *value) {return value;}

  public:
    virtual const char *getConfigValue(const char *key) const
        {return NULL;}
    virtual const KeyValue& getConfigEntry(const char *key) const
        {return nullKeyValue;}
    virtual const char *getPerObjectConfigValue(const char *objectFullPath,
        const char *keySuffix) const {return NULL;}
    virtual const KeyValue& getPerObjectConfigEntry(const char *objectFullPath,
        const char *keySuffix) const {return nullKeyValue;}
};
```

16.2.5 Loading NED files

NED files can be loaded with any of the following static methods of cSimulation: loadNedSourceFolder(), loadNedFile(), and loadNedText(). The first one loads a whole subdirectory tree, the second one loads a single NED file, and the third takes a literal string containing NED code and parses that one.

> **NOTE**
> One use of loadNedText() is to parse NED sources previously converted to C++ string constants and linked into the executable. This allows for creating executables that are self-contained, and do not need NED files to be distributed with them.

The above functions can be mixed as well, but after the last call, doneLoadingNedFiles() must be invoked (it checks for unresolved NED types).

Loading NED files has global effect, and they cannot be unloaded.

16.2.6 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 as well, albeit 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 your 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 OMNeT++ 3.x.

16.2.7 Assigning module parameters

As already mentioned, modules get 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 so that it can assign the parameter. For 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 there is one (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());
    }
};
```

16.2.8 Extracting statistics from the model

There are several ways you can extract statistics from the simulation.

Via 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 your 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 = simulation.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.

Via `cEnvir` callbacks

A more general way is to catch `recordScalar()` method calls in the simulation model. `cModule`'s `recordScalar()` method delegates to the similar function in `cEnvir`. You may define the latter function such 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 way.

A draft 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=NULL)
    {
       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 is that compile-time checking of statistics names is lost, but definite advantages include that any simulation model can now be used without changes, and that capturing additional statistics does not require code modification in the main program.

16.2.9 The simulation loop

To run the simulation, the `selectNextModule()` and `doOneEvent` methods of `cSimulation` must be called in a loop:

```
while (sim->getSimTime() < limit)
{
    cSimpleModule *mod = sim->selectNextModule();
    sim->doOneEvent(mod);
}
```

It depends on the concrete scheduler class whether `selectNextModule()` may return `NULL`. The default `cSequentialScheduler` never returns `NULL`.

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 CPU time limit, or when results reach a required accuracy. It is relatively straightforward to build in progress reporting and interactivity (start/stop) as well.

Animation can be hooked up to the appropriate callback methods of `cEnvir`: `beginSend()`, `sendHop()`, `endSend()`, and others.

16.2.10 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. This is necessary because several models and simulation kernel methods refer to the active simulation manager instance via the `simulation` macro, and it is similar with the `ev` macro.

> **NOTE**
> Before the 4.0 version, `simulation` and `ev` were global variables; now they are macros that resolve to `*cSimulation::getActiveSimulation()` and `*cSimulation::getActiveSimulation()->getEnvir()`.

Every `cSimulation` instance should have its own associated environment object (`cEnvir`). Environment objects may not be shared among several `cSimulation` instances. `cSimulation`'s destructor deletes the associated `cEnvir` instance as well.

`cSimulation` instances may be reused from one simulation to another, but it is also possible to make 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 get rid of, like the active simulation manager pointer and the active environment object pointer. Static buffers and objects (like string pools) are also used for efficiency reasons at some places inside the simulation kernel.

16.2.11 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();
simulation.setScheduler(scheduler);
```

It is usually not a good idea to change schedulers in the middle of a simulation, so `setScheduler()` may only be called when no network is set up.

16.2.12 Multi-threaded programs

The OMNeT++ simulation kernel is not reentrant, so it must be protected against concurrent access.

%

```
% $ ./fddi -h
%
% {\opp} Discrete Event Simulation  (C) 1992-2004 Andras Varga
% ...
% Available networks:
%   FDDI1
%   NRing
%   TUBw
%   TUBs
%
% Available modules:
%   FDDI_MAC
%   FDDI_MAC4Ring
%   ...
%
% Available channels:
%   ...
% End run of {\opp}
%
```

%

```
% Define_Module(FIFO);
%
```

%

```
% static cModule *FIFO__create(const char *name, cModule *parentmod)
% {
%     return new FIFO(name, parentmod);
% }
%
% EXECUTE_ON_STARTUP( FIFO__mod,
%     modtypes.getInstance()->add(
%        new cModuleType("FIFO","FIFO",(ModuleCreateFunc)FIFO__create)
%     );
% )
%
```

%

```
% cEnvir ev;
%
```

---

17 Appendix: NED Reference

17.1 Syntax

17.1.1 NED file extension

NED files have the `.ned` file name suffix. This is mandatory, and cannot be overridden.

17.1.2 NED file encoding

NED files are read as 8-bit-clean ASCII. This permits NED files saved in UTF-8, or any character set compatible with ASCII (8859-1, etc).

> **NOTE**
> There is no official way to determine the encoding of a NED file. It is up to the user to configure the correct encoding in text editors and other tools that are used to edit or process NED files.

Keywords and other syntactic elements in NED are ASCII, and identifiers must be ASCII as well. Comments and string literals may contain characters above 127. 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.

17.1.3 Reserved words

Authors have to take care that no reserved words are used as identifiers. The reserved words of the NED language are:

```
allowunconnected bool channel channelinterface connections const default double
extends false for gates if import index inout input int like module moduleinterface
network output package parameters property simple sizeof string submodules this true
types volatile xml xmldoc
```

17.1.4 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*).

17.1.5 Case sensitivity

Keywords and identifiers in the NED language are case sensitive. For example, `TCP` and `Tcp` are two different names.

17.1.6 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 their usual decimal or scientific notations.

Quantity constants

A quantity constant has the form *(<numeric-constant> <unit>)+*, for example `12.5mW` or `3h 15min 37.2s`. When multiple measurement units are present, they have to be convertible into each other (i.e. refer to the same physical quantity).

Section [17.5.7] lists the units recognized by OMNeT++. Other units can be used as well, the only downside being that OMNeT++ will not be able to perform conversions on them.

17.1.7 Comments

Comments can be placed anywhere in the NED file, with the usual C++ syntax: comments begin with a double slash `//', and last until the end of the line.

17.1.8 Grammar

The grammar of the NED language can be found in Appendix [18].

17.2 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.

```
package ned;

@namespace("");

channel IdealChannel {
    @class(cIdealChannel);
}

channel DelayChannel {
    @class(cDelayChannel);
    bool disabled = false;
    double delay = 0s @unit(s);
}

channel DatarateChannel {
    @class(cDatarateChannel);
    bool disabled = false;
    double delay = 0s @unit(s);
    double datarate = 0bps @unit(bps);
    double ber = 0;
    double per = 0;
}

moduleinterface IBidirectionalChannel {
    gates:
        inout a;
        inout b;
}

moduleinterface IUnidirectionalChannel {
    gates:
        input i;
        output o;
}
```

17.3 Packages

NED supports hierarchical namespaces called *packages*. The solution is roughly modelled after Java's packages, with minor changes.

17.3.1 Package declaration

The package of the declarations in a NED file is determined by the package declaration in the file (**`package`** keyword). A NED file may contain at most one package declaration. If there is no package declaration, the file's contents is in the *default package*.

Component type names must be unique within their package.

17.3.2 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*. The `package.ned` file directly in 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 to be in package `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, but they cannot be used to define the package they are in.

17.4 Components

Simple modules, compound modules, networks, channels, module interfaces and channel interfaces are called *components*.

17.4.1 Simple modules

Simple module types are declared with the **`simple`** keyword; see the NED Grammar (Appendix [18]) for the syntax.

Simple modules may have properties ([17.4.8]), parameters ([17.4.9]) and gates ([17.4.10]).

A simple module type may not have inner types ([17.4.13]).

A simple module type may extend another simple module type, and may implement one or more module interfaces ([17.4.5]). Inheritance rules are described in section [17.4.17], and interface implementation rules in section [17.4.16].

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 [17.4.7].

17.4.2 Compound modules

Compound module types are declared with the **`module`** keyword; see the NED Grammar (Appendix [18]) for the syntax.

Compound modules may have properties ([17.4.8]), parameters ([17.4.9]), and gates ([17.4.10]); its internal structure is defined by its submodules ([17.4.11]) and connections ([17.4.12]); and it may also have inner types ([17.4.13]) that can be used for submodules and connections.

A compound module type may extend another compound module type, and may implement one or more module interfaces ([17.4.5]). Inheritance rules are described in section [17.4.17], and interface implementation rules in section [17.4.16].

17.4.3 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 on 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 does not get 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.

17.4.4 Channels

Channel types are declared with the **`channel`** keyword; see the NED Grammar (Appendix [18]) for the syntax.

Channel types may have properties ([17.4.8]) and parameters ([17.4.9]).

A channel type may not have inner types ([17.4.13]).

A channel type may extend another channel type, and may implement one or more channel interfaces ([17.4.6]). Inheritance rules are described in section [17.4.17], and interface implementation rules in section [17.4.16].

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 [17.4.7].

17.4.5 Module interfaces

Module interface types are declared with the **`moduleinterface`** keyword; see the NED Grammar (Appendix [18]) for the syntax.

Module interfaces may have properties ([17.4.8]), parameters ([17.4.9]), and gates ([17.4.10]). However, parameters are not allowed to have a value assigned, not even a default value.

A module interface type may not have inner types ([17.4.13]).

A module interface type may extend one or more other module interface types. Inheritance rules are described in section [17.4.17].

17.4.6 Channel interfaces

Channel interface types are declared with the **`channelinterface`** keyword; see the NED Grammar (Appendix [18]) for the syntax.

Channel interfaces may have properties ([17.4.8]) and parameters ([17.4.9]). However, parameters are not allowed to have a value assigned, not even a default value.

A channel interface type may not have inner types ([17.4.13]).

A channel interface type may extend one or more other channel interface types. Inheritance rules are described in section [17.4.17].

17.4.7 Resolving the implementation C++ 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 [17.4.7]) 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**
> NED subclassing does not imply subclassing the C++ implementation! If you want to subclass a simple module or channel in NED as well as in C++, you explicitly need to specify the `@class` property, otherwise the derived simple module or channel will continue to use the C++ class from its super type.

Current namespace

The *current namespace* is the value of the first `@namespace` property found while searching the following order:

□□□ the current NED file

- the `package.ned` file in the current package
- the `package.ned` file of the parent 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.

17.4.8 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 `foo`, and a third with the index `bar`.

```
@statistic();
@statistic[foo]();
@statistic[bar]();
```

Property value

The value of the property is specified inside parentheses. The value consists of *key-valuelist* pairs, separated by semicolons; values 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:

```
@prop(value1,value2)
@prop(value1,value2;key1=value11,value12,value13)
```

Property values have a liberal syntax (see Appendix [18]). 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. To add a value with quotation marks included, enclose it in an extra layer of quotes: `"\"foo\""`.

Example:

```
@prop(marks=the ! mark, "the , mark", "the ; mark", other marks);
```

Placement

Properties may be added to NED files, component types, parameters, gates, submodules and connections. For the exact syntax, see Appendix [18].

When a component type extends another component type(s), properties are merged. This is described in section

[17.4.17].

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.

17.4.9 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.

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 [17.4.8]).

Accepted parameter data types are **double**, **int**, **string**, **bool**, and **xml**. Any of the above types can be declared **volatile** as well (`volatile int`, `volatile string`, etc.)

The presence of a data type keyword decides whether the given line defines a new parameter or refers to an existing parameter. One can assign 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 [17.5].

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 use within expressions to define constant subexpressions, i.e. to denote a part within an expression the should only be evaluated once. Constant subexpressions are not supported yet.

The following properties are recognized for parameters: `@unit`, `@prompt`.

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(s)
@unit("second")
@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) = 5;   // error: should be 5s
  double a @unit(s) = 5kg; // error: incompatible unit
```

@unit behavior for non-numeric parameters (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
}
```

17.4.10 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 [17.4.8]).

Gates may be scalar or vector. The vector size is specified with a numeric expression inside the square brackets (`[...]`). The vector size may also be left open by specifying an empty pair of square brackets.

A gate vector size may not be overridden in subclasses or in a submodule.

The presence of a gate type keyword decides 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 [17.5].

See the Connections section ([17.4.12]) for more information on gates.

Recognized gate properties

The following properties are recognized on 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 [17.4.12]).

`@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;
```

17.4.11 Submodules

Submodules are defined in the **`submodules`** section of the compound module.

Submodules may be scalar or vector, the vector size of the latter being specified with a numeric expression inside the square brackets (`[...]`).

Submodule type

The simple or compound module type ([17.4.1], [17.4.2]) that will be instantiated as the submodule may be specified in several ways:

- with a concrete module type name, or
- by a string-valued expression that evaluates to the name of a module type, or
- the type name may come from the configuration.

In the latter two cases, the **`like`** keyword is used, and a module interface type ([17.4.5]) needs to be specified as well, which the concrete module type needs to implement to be eligible to be chosen. In the second case, the string expression is specified in angle braces (`<...>`); in the third case, an empty pair of angle braces is used (<>). See the NED Grammar (Appendix [18]) for the exact syntax.

> **NOTE**
> When using the <> syntax, the actual NED type should be provided with the **`type-name`** configuration option: `**.host[0..3].type-name="MobileHost"`.

Parameters, gates

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.

A submodule body may contain parameters ([17.4.9]) and gates ([17.4.5]).

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 properties and parameter/gate properties.

17.4.12 Connections

Connections are defined in the **`connections`** section 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 [17.4.10].

Connections and 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 ([18]) 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;
}
```

Connection syntax

The connection syntax uses arrows (`-->`, `<--`) to connect **`input`** and **`output`** gates, and double arrows (`<-->`) to connect two **`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 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 the compound module, it is an error to use `++` on gate vector with no unconnected gates.

> **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 [17.4.4]) 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 sub-gates 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.go --> b.gi;
a.gi <-- b.go;
```

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 [17.4.4]).

The channel type that will be instantiated as channel object may be specified in several ways:

- with a concrete channel type name, or
- may be implicitly determined from the set of parameters used (see later), or
- by a string-valued expression that evaluates to the name of a channel type, or
- the type name may come from the configuration.

In the latter two cases, the **`like`** keyword is used, and a channel interface type ([17.4.6]) needs to be specified as

well, which the concrete channel type needs to implement to be eligible to be chosen. In the second case, the string expression is specified in angle braces (`<...>`); in the third case, an empty pair of angle braces is used (<>). The syntax is similar to the submodule syntax (see [17.4.11]).

> **NOTE**
> When using the <> syntax, the actual NED type should be provided with the **`type-name`** configuration option, where the object pattern should be the source gate plus `.channel`: `**.host[*].g$o.channel.type-name="EthernetChannel"`.

> **NOTE**
> As bidirectional connections (those using double arrows, `<-->`) are a shorthand for two uni-directional connections, bidirectional connections with channels will actually create *two* channel objects, one for each direction!

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 ([17.4.9]).

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.

Implicit channel type

When a connection uses an unnamed channel type (`--> {...} -->` syntax), the actual NED type to be used will depend on the parameters set in the connection.

When no parameters are set, `ned.IdealChannel` is chosen.

When only `ned.DelayChannel` parameters are used (`delay` and `disabled`), `ned.DelayChannel` is chosen.

When only `ned.DatarateChannel` parameters are used (`datarate`, `delay`, `ber`, `per`, `disabled`), the chosen channel type will be `ned.DatarateChannel`.

Unnamed channels connections cannot use any other parameters.

17.4.13 Inner types

Inner types are defined in the **`types`** section of the component type.

Inner types are only visible inside the enclosing component type.

17.4.14 Name uniqueness

Identifier names within a component must be unique. That is, submodule or inner type cannot be named the same, or the same as a gate or parameter of the parent module.

17.4.15 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;
- 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.protocols.network.ip.RoutingTable;
import inet.protocols.network.ip.*;
import inet.protocols.network.ip.Ro*Ta*;
import inet.protocols.*.ip.*;
import inet.**.RoutingTable;
```

One asterisk "*" stands for "any character sequence not containing period"; two asterisks mean "any character sequence which may contain period". No other wildcards are recognized.

An import not containing 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.

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.

Parametric module types ("like" submodules)

Lookup of the actual module type for "like" submodules differs from normal lookups. This lookup ignores the imports in the file altogether. Instead, it collects all modules that support the given interface and match the given type name string (i.e. end in the same simple name, or have the same fully qualified name). The result must be exactly one module type.

The algorithm for parametric channel types works in the same way.

Network name in the ini file

The network name in the ini file may be given as fully qualified name or as simple (unqualified) name.

Simple (unqualified) names are tried with the same package as the ini file is in (provided it's in a NED directory).

17.4.16 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 those properties, with at least the same values. It 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.

17.4.17 Inheritance

- A simple module may only extend a simple module.
- A compound module may only extend a compound 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 redefined in subclasses or at place of usage
- for gate/parameter properties: extra properties can be added; existing properties can be overridden/extended like 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

See other rules for specifics.

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 the special value '-')

Parameter inheritance

Default values for parameters may be overridden in subclasses.

Parameter (non-default) assignments may also be overridden in subclasses.

Rationale
The latter is needed for ease of use of channels and their `delay`, `ber`, `per`, `datarate` parameters.

Gate inheritance

Gate vector size may not be overridden in subclasses.

17.4.18 Network build order

When a network in 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 get 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.

17.5 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.

17.5.1 Operators

The following operators are supported (in order of decreasing precedence):

Operator	Meaning
`-`, `!`, `~`	unary minus, negation, bitwise complement
`^`	power-of
`*`, `/`, `%`	multiply, divide, modulus
`+`, `-`	add, subtract, string concatenation
`<<`, `>>`	bitwise shift
`&`, `|`, `#`	bitwise and, or, xor
`==`	equal
`!=`	not equal
`>`, `>=`	greater, greater or equal
`<`, `<=`	less, less or equal
`&&`, `||`, `##`	logical operators and, or, xor
`?:`	the C/C++ ``inline if''

Conversions

Values are of type boolean, long, double, string, or xml element. A long or double may have an associated measurement unit (`s`, `mW`, etc.)

Long and double values are implicitly converted to one another where needed. Conversion is performed with the C++ language's built-in typecast operator.

There is no implicit conversion between bool 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 vica versa.

> **NOTE**
> Implementation note: Currently the stack-based evaluation engine represents all numbers in `double`, i.e. it does not use `long` on the stack. (This may change in future releases.) Details:
>
> - Integers (integer constants, `long` parameters, etc) are converted to `double`.
> - For bitwise operators, `doubles` are converted to `unsigned long` using the C/C++ built-in conversion (type cast), the operation is performed, then the result is converted back to `double`.
> - For modulus (%), the operands are converted to `long`.

Unit handling

Operations involving numbers with units work in the following way.

Addition, substraction, 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.

Power-of and bitwise operations require their arguments to be dimensionless.

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.

17.5.2 Referencing parameters and loop variables

Identifiers in expressions occurring *anywhere* in component definitions are interpreted as referring to parameters of the given component.

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.

In submodule bodies, parameters of the same submodule can be referred to with the **`this`** qualifier: `this.destAddress`.

Expressions may also refer to parameters of submodules defined earlier in NED file, using the `submoduleName.paramName` or the `submoduleName[index].paramName` syntax.

17.5.3 The `index` operator

The **`index`** operator is only allowed in a vector submodule's body, and yields the index of the submodule instance.

17.5.4 The `sizeof()` operator

The **`sizeof()`** operator expects one argument, and it is only accepted in compound module definitions.

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.

17.5.5 The `xmldoc()` operator

The **`xmldoc()`** operator can be used to assign XML-type parameters, that is, point them to XML files or to specific elements inside XML files.

`xmldoc()` has two flavours: one accepts a file name, the second accepts a file name plus an XPath-like expression which selects an element inside the XML file.

`xmldoc()` with two arguments accepts a path expression to select an element within the document. The expression syntax is similar to XPath.

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 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`.

17.5.6 Functions

The functions available in NED are listed in Appendix [20].

17.5.7 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
s	second	
d	day	86400s
h	hour	3600s
min	minute	60s
ms	millisecond	1e-3s
us	microsecond	1e-6s
ns	nanosecond	1e-9s

ps	picosecond	1e-12s
bps	bit/sec	
Kbps	kilobit/sec	1e3bps
Mbps	megabit/sec	1e6bps
Gbps	gigabit/sec	1e9bps
Tbps	terabit/sec	1e12bps
B	byte	
KB	kilobyte	1024B
MB	megabyte	1.04858e6B
GB	gigabyte	1.07374e9B
TB	terabyte	1.09951e12B
b	bit	
m	meter	
km	kilometer	1e3m
cm	centimeter	1e-2m
mm	millimeter	1e-3m
W	watt	
mW	milliwatt	1e-3W
Hz	herz	
kHz	kiloherz	1e3Hz
MHz	megaherz	1e6Hz
GHz	gigaherz	1e9Hz
g	gram	1e-3kg
kg	kilogram	
J	joule	
kJ	kilojoule	1e3J
MJ	megajoule	1e6J
V	volt	
kV	kilovolt	1e3V
mV	millivolt	1e-3V
A	amper	
mA	milliamper	1e-3A
uA	microamper	1e-6A

---

18 Appendix: 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*/*yacc*
- 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)
- other terminals represent keywords in all lowercase

```
nedfile
        : definitions
        |
        ;

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 '@' NAME
        | PROPERTY '@' NAME '[' ']'
        ;

opt_propertydecl_keys
        : propertydecl_keys
        |
        ;

propertydecl_keys
        : propertydecl_keys ';' propertydecl_key
        | propertydecl_key
        ;

propertydecl_key
        : NAME
        ;

fileproperty
        : property_namevalue ';'
        ;
```

```
channeldefinition
        : channelheader '{'
            opt_paramblock
          '}'
        ;


channelheader
        : CHANNEL NAME
          opt_inheritance
        ;


opt_inheritance
        :
        | 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
        |
```

```
        ;

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
      |
      ;


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
        | XML
        ;

opt_volatile
        : VOLATILE
        |
        ;

paramvalue
        : expression
        | DEFAULT '(' expression ')'
        | DEFAULT
        | ASK
        ;

opt_inline_properties
        : inline_properties
        |
        ;

inline_properties
        : inline_properties property_namevalue
        | property_namevalue
        ;

pattern
        : pattern pattern_elem
        | pattern_elem
        ;
```

```
pattern_elem
        : '.'
        | '*'
        | '?'
        | '**'
        | NAME
        | INTCONSTANT
        | '..'
        | '[' pattern ']'
        | '{' pattern '}'
        | IMPORT | PACKAGE | PROPERTY
        | MODULE | SIMPLE | NETWORK | CHANNEL | MODULEINTERFACE | CHANNELINTERFACE
        | EXTENDS | LIKE
        | DOUBLE | INT | STRING | BOOL | XML | VOLATILE
        | INPUT | OUTPUT | INOUT | IF | FOR
        | TYPES | PARAMETERS | GATES | SUBMODULES | CONNECTIONS | ALLOWUNCONNECTED
        | TRUE | FALSE | THIS | DEFAULT | CONST | SIZEOF | INDEX | XMLDOC
        ;


property
        : property_namevalue ';'
        ;


property_namevalue
        : property_name
        | property_name '(' opt_property_keys ')'
        ;


property_name
        : '@' NAME
        | '@' NAME '[' NAME ']'
        ;


opt_property_keys
        : property_keys
        ;


property_keys
        : property_keys ';' property_key
        | property_key
        ;


property_key
        : NAME '=' property_values
        | property_values
        ;
```

```
property_values
        : property_values ',' property_value
        | property_value
        ;


property_value
        : property_value_tokens
        | STRINGCONSTANT
        |
        ;


property_value_tokens
        : property_value_tokens property_value_token
        | property_value_token
        ;


property_value_token
        : NAME | INTCONSTANT | REALCONSTANT | TRUE | FALSE
        | '$' | '@' | ':' | '=' | '[' | ']' | '{' | '}' | '.' | '?'
        | '^' | '+' | '-' | '*' | '/' | '%' | '<' | '>' | '==' | '!=' | '<=' | '>='
        | '**' | '..' | '++' | '||' | '&&' | '##' | '!'
        ;


opt_gateblock
        : gateblock
        |
        ;


gateblock
        : GATES ':'
          opt_gates
        ;


opt_gates
        : gates
        |
        ;


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
        |
        ;


typeblock
        : TYPES ':'
          opt_localtypes
        ;


opt_localtypes
        : localtypes
        |
        ;


localtypes
        : localtypes localtype
        | localtype
        ;


localtype
        : propertydecl
        | channeldefinition
        | channelinterfacedefinition
        | simplemoduledefinition
        | compoundmoduledefinition
```

```
        | networkdefinition
        | moduleinterfacedefinition
        | ';'
        ;


opt_submodblock
        : submodblock
        |
        ;


submodblock
        : SUBMODULES ':'
          opt_submodules
        ;


opt_submodules
        : submodules
        |
        ;


submodules
        : submodules submodule
        | submodule
        ;


submodule
        : submoduleheader ';'
        | submoduleheader '{'
          opt_paramblock
          opt_gateblock
          '}' opt_semicolon
        ;


submoduleheader
        : submodulename ':' dottedname
        | submodulename ':' likeparam LIKE dottedname
        ;


submodulename
        : NAME
        |  NAME vector
        ;


likeparam
        : '<' '>'
```

```
        | '<' expression '>'
        ;


opt_connblock
        : connblock
        |
        ;


connblock
        : CONNECTIONS ALLOWUNCONNECTED ':'
          opt_connections
        | CONNECTIONS ':'
          opt_connections
        ;


opt_connections
        : connections
        |
        ;


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
        |
        ;


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
        |
        ;


channelspec
        : channelspec_header
        | channelspec_header '{'
            opt_paramblock
          '}'
        ;


channelspec_header
        :
        | dottedname
        | likeparam LIKE dottedname
        ;


condition
        : IF expression
        ;


vector
        : '[' expression ']'
        ;


expression
        :
```

```
        expr
      | xmldocvalue
      ;


xmldocvalue
      : XMLDOC '(' stringliteral ',' stringliteral ')'
      | XMLDOC '(' stringliteral ')'
      ;


expr
      : simple_expr
      | '(' expr ')'
      | CONST '(' 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
      | INT '(' expr ')'
      | DOUBLE '(' expr ')'
      | STRING '(' expr ')'
      | NAME '(' ')'
      | NAME '(' expr ')'
      | NAME '(' expr ',' expr ')'
      | NAME '(' expr ',' expr ',' expr ')'
      | NAME '(' expr ',' expr ',' expr ',' expr ')'
      | NAME '(' expr ',' expr ',' expr ',' expr ',' expr ')'
      | NAME '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ')'
      | NAME '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ')'
      | NAME '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ')'
```

```
        | NAME '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ')'
        | NAME '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ')'
        ;


simple_expr
        : identifier
        | special_expr
        | literal
        ;


identifier
        : NAME
        | THIS '.' NAME
        | NAME '.' NAME
        | NAME '[' expr ']' '.' NAME
        ;


special_expr
        : INDEX
        | INDEX '(' ')'
        | SIZEOF '(' identifier ')'
        ;


literal
        : stringliteral
        | boolliteral
        | numliteral
        ;


stringliteral
        : STRINGCONSTANT
        ;


boolliteral
        : TRUE
        | FALSE
        ;


numliteral
        : INTCONSTANT
        | REALCONSTANT
        | quantity
        ;


quantity
```

```
    : quantity INTCONSTANT NAME
    | quantity REALCONSTANT NAME
    | INTCONSTANT NAME
    | REALCONSTANT NAME
    ;


opt_semicolon
    : ';'
    |
    ;
```

---

19 Appendix: NED XML Binding

This appendix shows the DTD for the XML binding of the NED language and message definitions.

```
<!ELEMENT files ((ned-file|msg-file)*)>

<!--
  **  NED-2.
  -->

<!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                NMTOKEN   #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|pattern)*)>
<!ATTLIST parameters
    is-implicit         (true|false)  "false">

<!ELEMENT param (comment*, expression?, property*)>
<!ATTLIST param
     type               (double|int|string|bool|xml) #IMPLIED
     is-volatile        (true|false)  "false"
     name               NMTOKEN   #REQUIRED
     value              CDATA     #IMPLIED
     is-default         (true|false)  "false">

<!ELEMENT pattern (comment*, expression?, property*)>
<!ATTLIST pattern
     pattern            CDATA     #REQUIRED
     value              CDATA     #IMPLIED
     is-default         (true|false)  "false">

<!ELEMENT property (comment*, property-key*)>
<!ATTLIST property
     is-implicit        (true|false) "false"
     name               NMTOKEN   #REQUIRED
     index              NMTOKEN   #IMPLIED>

<!ELEMENT property-key (comment*, literal*)>
<!ATTLIST property-key
     name               CDATA     #IMPLIED>

<!ELEMENT gates (comment*, gate*)>

<!ELEMENT gate (comment*, expression?, 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*, expression*, parameters?, gates?)>
```

```
<!ATTLIST submodule
     name               NMTOKEN   #REQUIRED
     type               CDATA     #IMPLIED
     like-type          CDATA     #IMPLIED
     like-param         CDATA     #IMPLIED
     vector-size        CDATA     #IMPLIED>

<!ELEMENT connections (comment*, (connection|connection-group)*)>
<!ATTLIST connections
     allow-unconnected (true|false) "false">

<!ELEMENT connection (comment*, expression*, channel-spec?, (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
     arrow-direction    (l2r|r2l|bidir) #REQUIRED>

<!ELEMENT channel-spec (comment*, expression*, parameters?)>
<!ATTLIST channel-spec
     type               CDATA     #IMPLIED
     like-type          CDATA     #IMPLIED
     like-param         CDATA     #IMPLIED>

<!ELEMENT connection-group (comment*, (loop|condition)*, connection*)>

<!ELEMENT loop (comment*, expression*)>
<!ATTLIST loop
     param-name          NMTOKEN   #REQUIRED
     from-value          CDATA     #IMPLIED
     to-value            CDATA     #IMPLIED>

<!ELEMENT condition (comment*, expression?)>
<!ATTLIST condition
     condition           CDATA     #IMPLIED>

<!--
  **  Expressions
  -->

<!ELEMENT expression (comment*, (operator|function|ident|literal))>
<!ATTLIST expression
     target              CDATA     #IMPLIED>

<!ELEMENT operator (comment*, (operator|function|ident|literal)+)>
<!ATTLIST operator
     name                CDATA     #REQUIRED>

<!-- functions, "index", "const" and "sizeof" -->
<!ELEMENT function (comment*, (operator|function|ident|literal)*)>
<!ATTLIST function
     name                NMTOKEN   #REQUIRED>

<!-- Ident is either a parameter reference or an identifier for 'sizeof' operator
     format is 'name' or 'module.name' or 'module[n].name'. If there's a child,
     that's the module index n.
-->
<!ELEMENT ident (comment*, (operator|function|ident|literal)?)>
<!ATTLIST ident
     module              CDATA     #IMPLIED
```

```
    name                NMTOKEN   #REQUIRED>


<!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. Unit can only be present with "double".
 -->
<!ATTLIST literal
    type  (double|int|string|bool|spec)  #REQUIRED
    unit                CDATA     #IMPLIED
    text                CDATA     #IMPLIED
    value               CDATA     #IMPLIED>

<!--============================================================-->

<!--
**
** OMNeT++/OMNEST Message Definitions (MSG)
**
-->

<!ELEMENT msg-file (comment*, (namespace|property-decl|property|cplusplus|
                   struct-decl|class-decl|message-decl|packet-decl|enum-decl|
                   struct|class|message|packet|enum)*)>

<!ATTLIST msg-file
    filename            CDATA     #IMPLIED
    version             CDATA     "2">

<!ELEMENT namespace (comment*)>
<!ATTLIST namespace
    name                CDATA     #REQUIRED>  <!-- note: not NMTOKEN because it
may contain "::" -->

<!ELEMENT cplusplus (comment*)>
<!ATTLIST cplusplus
    body                CDATA     #REQUIRED>

<!-- C++ type announcements -->

<!ELEMENT struct-decl (comment*)>
<!ATTLIST struct-decl
    name                NMTOKEN   #REQUIRED>

<!ELEMENT class-decl (comment*)>
<!ATTLIST class-decl
    name                NMTOKEN   #REQUIRED
    is-cobject      (true|false)  "false"
    extends-name        NMTOKEN   #IMPLIED>

<!ELEMENT message-decl (comment*)>
<!ATTLIST message-decl
    name                NMTOKEN   #REQUIRED>

<!ELEMENT packet-decl (comment*)>
<!ATTLIST packet-decl
    name                NMTOKEN   #REQUIRED>

<!ELEMENT enum-decl (comment*)>
<!ATTLIST enum-decl
    name                NMTOKEN   #REQUIRED>

<!-- Enums -->

<!ELEMENT enum (comment*, enum-fields?)>
<!ATTLIST enum
    name                NMTOKEN   #REQUIRED
```

```
     extends-name         NMTOKEN    #IMPLIED
     source-code          CDATA      #IMPLIED>

<!ELEMENT enum-fields (comment*, enum-field*)>

<!ELEMENT enum-field (comment*)>
<!ATTLIST enum-field
     name                 NMTOKEN    #REQUIRED
     value                CDATA      #IMPLIED>

<!-- Message, class, struct -->

<!ELEMENT message (comment*, (property|field)*)>
<!ATTLIST message
     name                 NMTOKEN    #REQUIRED
     extends-name         NMTOKEN    #IMPLIED
     source-code          CDATA      #IMPLIED>

<!ELEMENT packet (comment*, (property|field)*)>
<!ATTLIST packet
     name                 NMTOKEN    #REQUIRED
     extends-name         NMTOKEN    #IMPLIED
     source-code          CDATA      #IMPLIED>

<!ELEMENT class (comment*, (property|field)*)>
<!ATTLIST class
     name                 NMTOKEN    #REQUIRED
     extends-name         NMTOKEN    #IMPLIED
     source-code          CDATA      #IMPLIED>

<!ELEMENT struct (comment*, (property|field)*)>
<!ATTLIST struct
     name                 NMTOKEN    #REQUIRED
     extends-name         NMTOKEN    #IMPLIED
     source-code          CDATA      #IMPLIED>

<!ELEMENT field (comment*)>
<!ATTLIST field
     name                 NMTOKEN    #REQUIRED
     data-type            CDATA      #IMPLIED
     is-abstract      (true|false)   "false"
     is-readonly      (true|false)   "false"
     is-vector        (true|false)   "false"
     vector-size          CDATA      #IMPLIED
     enum-name            NMTOKEN    #IMPLIED
     default-value        CDATA      #IMPLIED>

<!--
  **  'unknown' is used internally to represent elements not in this NED DTD
  -->
<!ELEMENT unknown         ANY>
<!ATTLIST unknown
     element              CDATA      #REQUIRED>
```

---

20 Appendix: 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.

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.

- [int]: `long int(any x)`

Converts x to long, and returns the result. A boolean argument becomes 0 or 1; a double is converted using floor(); a string is interpreted as number; an XML argument causes an error.

- [string]: `string string(any x)`

Converts x to string, and returns the result.

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 down; 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 up; 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

Category "ned":

- [ancestorIndex]: `long ancestorIndex(long 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]: `long parentIndex()`

Returns the index of the parent module, which has to be part of module vector.

Category "random/continuous":

- [beta]: `double beta(double alpha1, double alpha2, long rng?)`

Returns a random number from the Beta distribution

- [cauchy]: `quantity cauchy(quantity a, quantity b, long rng?)`

Returns a random number from the Cauchy distribution

- [chi_square]: `double chi_square(long k, long rng?)`

Returns a random number from the Chi-square distribution

- [erlang_k]: `quantity erlang_k(long k, quantity mean, long rng?)`

Returns a random number from the Erlang distribution

- [exponential]: `quantity exponential(quantity mean, long rng?)`

Returns a random number from the Exponential distribution

- [gamma_d]: `quantity gamma_d(double alpha, quantity theta, long rng?)`

Returns a random number from the Gamma distribution

- [lognormal]: `double lognormal(double m, double w, long rng?)`

Returns a random number from the Lognormal distribution

- [normal]: `quantity normal(quantity mean, quantity stddev, long rng?)`

Returns a random number from the Normal distribution

- [pareto_shifted]: `quantity pareto_shifted(double a, quantity b, quantity c, long rng?)`

Returns a random number from the Pareto-shifted distribution

- [student_t]: `double student_t(long i, long rng?)`

Returns a random number from the Student-t distribution

- [triang]: `quantity triang(quantity a, quantity b, quantity c, long rng?)`

Returns a random number from the Triangular distribution

- [truncnormal]: `quantity truncnormal(quantity mean, quantity stddev, long rng?)`

Returns a random number from the Truncated Normal distribution

- [uniform]: `quantity uniform(quantity a, quantity b, long rng?)`

Returns a random number from the Uniform distribution

- [weibull]: `quantity weibull(quantity a, quantity b, long rng?)`

Returns a random number from the Weibull distribution

Category "random/discrete":

- [bernoulli]: `long bernoulli(double p, long rng?)`

Returns a random number from the Bernoulli distribution

- [binomial]: `long binomial(long n, double p, long rng?)`

Returns a random number from the Binomial distribution

- [geometric]: `long geometric(double p, long rng?)`

Returns a random number from the Geometric distribution

- [intuniform]: `long intuniform(long a, long b, long rng?)`

Returns a random number from the Intuniform distribution

- [negbinomial]: `long negbinomial(long n, double p, long rng?)`

Returns a random number from the Negbinomial distribution

- [poisson]: `long poisson(double lambda, long rng?)`

Returns a random number from the Poisson distribution

Category "strings":

- [choose]: `string choose(long 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.

- [indexOf]: `long 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]: `long length(string s)`

Returns the length of the string

- [replace]: `string replace(string s, string substr, string repl, long 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, long 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, long pos, long 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, long 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.

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.

---

21 Appendix: 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*/*yacc*
- 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 the previous versions of OMNeT++ (3.x) can be parsed.

```
msgfile
        : definitions
        ;

definitions
      : definitions definition
      |
      ;


definition
      : namespace_decl
      | cplusplus
      | struct_decl
      | class_decl
      | message_decl
      | packet_decl
      | enum_decl
      | enum
      | message
      | packet
      | class
      | struct
      ;


namespace_decl
      : NAMESPACE namespacename ';'


namespacename
      : namespacename ':' ':' NAME
      | NAME
      ;
```

```
cplusplus
        : CPLUSPLUS '{{' ... '}}' opt_semicolon
        ;


struct_decl
        : STRUCT NAME ';'
        ;


class_decl
        : CLASS NAME ';'
        | CLASS NONCOBJECT NAME ';'
        | CLASS NAME EXTENDS NAME ';'
        ;


message_decl
        : MESSAGE NAME ';'
        ;


packet_decl
        : PACKET NAME ';'
        ;


enum_decl
        : ENUM NAME ';'
        ;


enum
        : ENUM NAME '{'
          opt_enumfields '}' opt_semicolon
        | ENUM NAME EXTENDS NAME '{'
          opt_enumfields '}' opt_semicolon
        ;


opt_enumfields
        : enumfields
        |
        ;


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 NAME '{'
        | MESSAGE NAME EXTENDS NAME '{'
        ;

packet_header
        : PACKET NAME '{'
        | PACKET NAME EXTENDS NAME '{'
        ;

class_header
        : CLASS NAME '{'
        | CLASS NAME EXTENDS NAME '{'
        ;

struct_header
        : STRUCT NAME '{'
        | STRUCT NAME EXTENDS NAME '{'
        ;

body
        : opt_fields_and_properties
          opt_propertiesblock_old
          opt_fieldsblock_old
```

```
        '}' opt_semicolon
      ;


opt_fields_and_properties
      : fields_and_properties
      |
      ;


fields_and_properties
      : fields_and_properties field
      | fields_and_properties property
      | field
      | property
      ;


field
      : fieldmodifiers fielddatatype NAME
        opt_fieldvector opt_fieldenum opt_fieldvalue ';'
      | fieldmodifiers NAME
        opt_fieldvector opt_fieldenum opt_fieldvalue ';'
      ;


fieldmodifiers
      : ABSTRACT
      | READONLY
      | ABSTRACT READONLY
      | READONLY ABSTRACT
      |
      ;


fielddatatype
      : NAME
      | NAME '*'
      | CHAR
      | SHORT
      | INT
      | LONG
      | UNSIGNED CHAR
      | UNSIGNED SHORT
      | UNSIGNED INT
      | UNSIGNED LONG
      | DOUBLE
      | STRING
      | BOOL
      ;
```

```
opt_fieldvector
        : '[' INTCONSTANT ']'
        | '[' NAME ']'
        | '[' ']'
        |
        ;


opt_fieldenum
        : ENUM '(' NAME ')'
        |
        ;


opt_fieldvalue
        : '=' fieldvalue
        |
        ;


fieldvalue
        : fieldvalue fieldvalueitem
        | fieldvalueitem
        ;


fieldvalueitem
        : STRINGCONSTANT
        | CHARCONSTANT
        | INTCONSTANT
        | REALCONSTANT
        | TRUE
        | FALSE
        | NAME
        | '?' | ':' | '&&' | '||' | '##' | '==' | '!=' | '>' | '>=' | '<' | '<='
        | '&' | '|' | '#' | '<<' | '>>'
        | '+' | '-' | '*' | '/' | '%' | '^' | '&' | UMIN | '!' | '~'
        | '.' | ',' | '(' | ')' | '[' | ']'
        ;


enumvalue
        : INTCONSTANT
        | '-' INTCONSTANT
        | NAME
        ;


quantity
        : quantity INTCONSTANT NAME
        | quantity REALCONSTANT NAME
        | INTCONSTANT NAME
```

```
        | REALCONSTANT NAME
        ;


property
        : property_namevalue ';'
        ;


property_namevalue
        : property_name
        | property_name '(' opt_property_keys ')'
        ;


property_name
        : '@' NAME
        | '@' NAME '[' NAME ']'
        ;


opt_property_keys
        : property_keys
        ;


property_keys
        : property_keys ';' property_key
        | property_key
        ;


property_key
        : NAME '=' property_values
        | property_values
        ;


property_values
        : property_values ',' property_value
        | property_value
        ;


property_value
        : NAME
        | '$' NAME
        | STRINGCONSTANT
        | TRUE
        | FALSE
        | INTCONSTANT
        | REALCONSTANT
        | quantity
```

```
        | '-'
        |
        ;


opt_fieldsblock_old
        : FIELDS ':'
          opt_fields_old
        |
        ;


opt_fields_old
        : fields_old
        |
        ;


fields_old
        : fields_old field
        | field
        ;


opt_propertiesblock_old
        : PROPERTIES ':'
          opt_properties_old
        |
        ;


opt_properties_old
        : properties_old
        |
        ;


properties_old
        : properties_old property_old
        | property_old
        ;


property_old
        : NAME '=' property_value ';'
        ;


opt_semicolon : ';' | ;
```

22 Appendix: Display String Tags

22.1 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
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 (colorname or #RRGGBB or @HHSSBB). Default: #8080ff
b[4] - border color	Border color of the object (colorname or #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 color	A color to colorize the icon (colorname or #RRGGBB or @HHSSBB)
i[2] - icon colorization %	Amount of colorization 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 color	A color to colorize the overlay icon (colorname or #RRGGBB or @HHSSBB)
i2[2] - overlay icon colorization %	Amount of colorization in percent. Default: 30
r[0] - range	Radius of the range indicator
r[1] - range fill color	Fill color of the range indicator (colorname or #RRGGBB or @HHSSBB)
r[2] - range border color	Border color of the range indicator (colorname or #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 (colorname or #RRGGBB or @HHSSBB). Default: blue
tt[0] - tooltip	Tooltip to be displayed over the object
bgp[0] - bg x	Module background horizontal offset
bgp[1] - bg y	Module background vertical offset
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 (colorname or #RRGGBB or @HHSSBB). Default: grey82
bgb[3] - bg border color	Border color of the module background rectangle (colorname or #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 tick distance	Distance between two major ticks measured in units
bgg[1] - grid minor ticks	Minor ticks per major ticks. Default: 1
bgg[2] - grid color	Color of the grid lines (colorname or #RRGGBB or @HHSSBB). Default: grey
bgl[0] - layout seed	Seed value for layout algorithm
bgl[1] - layout algorithm	Algorithm for child layouting
bgs[0] - pixels per unit	Number of pixels per distance unit
bgs[1] - unit name	Name of distance unit
ls[0] - line color	Connection color (colorname or #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

22.2 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. For color notation, see section [10.3]. 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.

23 Appendix: Configuration Options

23.1 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.

```
cmdenv-autoflush=<bool>, default:false; per-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
    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.

<object-full-path>.cmdenv-ev-output=<bool>, default:true; per-object setting
    When cmdenv-express-mode=false: whether Cmdenv should print debug messages
    (ev<<) from the selected modules.

cmdenv-event-banner-details=<bool>, default:false; per-run setting
    When cmdenv-express-mode=false: print extra information after event
    banners.

cmdenv-event-banners=<bool>, default:true; per-run setting
    When cmdenv-express-mode=false: turns printing event banners on/off.

cmdenv-express-mode=<bool>, default:true; per-run setting
    Selects ``normal'' (debug/trace) or ``express'' mode.

cmdenv-extra-stack=<double>, unit="B", default:8KB; global setting
    Specifies the extra amount of stack that is reserved for each activity()
    simple module when the simulation is run under Cmdenv.

cmdenv-interactive=<bool>, default:false; global 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-message-trace=<bool>, default:false; per-run setting
    When cmdenv-express-mode=false: print a line per message sending (by
    send(),scheduleAt(), etc) and delivery on the standard output.

cmdenv-module-messages=<bool>, default:true; per-run setting
    When cmdenv-express-mode=false: turns printing module ev<< output on/off.

cmdenv-output-file=<filename>; global setting
    When a filename is specified, Cmdenv redirects standard output into the
    given file. This is especially useful with parallel simulation. See the
    `fname-append-host' option as well.

cmdenv-performance-display=<bool>, default:true; per-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-runs-to-execute=<string>; global setting
    Specifies which runs to execute from the selected configuration (see
    cmdenv-config-name=). It accepts 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=<int>, default:100000; per-run setting
    When cmdenv-express-mode=true: print status update every n events. Typical
    values are 100,000...1,000,000.

configuration-class=<string>; global setting
    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
    cConfiguration interface.

constraint=<string>; per-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-run setting
    Stops the simulation when CPU usage has reached the given limit. The
    default is no limit.

debug-on-errors=<bool>, default:false; global 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.

description=<string>; per-run setting
    Descriptive name for the given simulation configuration. Descriptions get
    displayed in the run selection dialog.

eventlog-file=<filename>, default:${resultdir}/${configname}-${runnumber}.elog;
per-run setting
    Name of the event log file to generate.

eventlog-message-detail-pattern=<custom>; per-run setting
    A list of patterns separated by '|' character which will be used to write
    message detail information into the event log 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 (enter them without quotes):
      "*": 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-recording-intervals=<custom>; per-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..

experiment-label=<string>, default:${configname}; per-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-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-run setting
    The expected fingerprint of the simulation. When provided, a fingerprint
    will be calculated from the simulation event times and other quantities
    during simulation, and checked against the given one. Fingerprints are
    suitable for crude regression tests. As fingerprints occasionally differ
    across platforms, more than one fingerprint values can be specified here,
    separated by spaces, and a match with any of them will be accepted. To
    calculate the initial fingerprint, enter any dummy string (such as "none"),
    and run the simulation.

fname-append-host=<bool>, default:false; global setting
    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.

load-libs=<filenames>; global setting
    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-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-run setting
    Identifies the measurement within the experiment. This string gets recorded
    into result files, and may be referred to during result analysis.

<object-full-path>.module-eventlog-recording=<bool>, default:true; per-object
setting
    Enables recording events on a per module basis. This is meaningful for
    simple modules only.
    Example:
     **.router[10..20].**.module-eventlog-recording = true
     **.module-eventlog-recording = false

ned-path=<path>; global setting
    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
    {\opp} IDE sets the NED path automatically, and for simulations started
    outside the IDE it is more convenient to specify it via a command-line
    option or the NEDPATH environment variable.

network=<string>; per-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-run setting
    The number of random number generators.

output-scalar-file=<filename>, default:${resultdir}/${configname}-
${runnumber}.sca; per-run setting
    Name for the output scalar file.

output-scalar-file-append=<bool>, default:false; per-run setting
    What to do when the output scalar file already exists: append to it
    ({\opp} 3.x behavior), or delete it and begin a new file (default).

output-scalar-precision=<int>, default:14; per-run setting
    The number of significant digits for recording data into the output scalar
    file. The maximum value is ~15 (IEEE double precision).
```

```
output-vector-file=<filename>, default:${resultdir}/${configname}-
${runnumber}.vec; per-run setting
    Name for the output vector file.

output-vector-precision=<int>, default:14; per-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 the "time" column of output vectors, which are represented as
    fixed-point numbers and always get recorded precisely.

output-vectors-memory-limit=<double>, unit="B", default:16MB; per-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.

outputscalarmanager-class=<string>, default:cFileOutputScalarManager; global
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 cOutputScalarManager interface.

outputvectormanager-class=<string>, default:cIndexedFileOutputVectorManager;
global 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 cOutputVectorManager interface.

parallel-simulation=<bool>, default:false; global setting
    Enables parallel distributed simulation.

<object-full-path>.param-record-as-scalar=<bool>, default:false; per-object
setting
    Applicable to module parameters: specifies whether the module parameter
    should be recorded into the output scalar file. Set it for parameters whose
    value you'll need for result analysis.

parsim-communications-class=<string>, default:cFileCommunications; global setting
    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
    With parallel-simulation=true: turns on printing of log messages from the
    parallel simulation code.

parsim-filecommunications-prefix=<string>, default:comm/; global setting
    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
    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
    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
    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
    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:omnetpp; global setting
    When cNamedPipeCommunications is selected as parsim communications class:
    selects the name prefix for Windows named pipes created.

parsim-nullmessageprotocol-laziness=<double>, default:0.5; global setting
    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
    When cNullMessageProtocol is selected as parsim synchronization class:
    specifies the C++ class that calculates lookahead. The class should
    subclass from cNMPLookahead.

parsim-synchronization-class=<string>, default:cNullMessageProtocol; global
setting
    If parallel-simulation=true, it selects the parallel simulation algorithm.
    The class must implement the cParsimSynchronizer interface.

<object-full-path>.partition-id=<string>; per-object setting
    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; global setting
    Whether to report objects left (that is, not deallocated by simple module
    destructors) after network cleanup.

realtimescheduler-scaling=<double>; global setting
    When cRealTimeScheduler is selected as scheduler class: ratio of simulation
    time to real time. For example, scaling=2 will cause simulation time to
    progress twice as fast as runtime.

record-eventlog=<bool>, default:false; per-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-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-run setting
    Identifies one replication of a measurement (see repeat= and
    measurement-label= as well). This string gets recorded into result files,
    and may be referred to during result analysis.

result-dir=<string>, default:results; per-run setting
    Value for the ${resultdir} variable, which is used as the default directory
    for result files (output vector file, output scalar file, eventlog file,
    etc.)

<object-full-path>.rng-%=<int>; per-object setting
    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
    default is one-to-one mapping.

rng-class=<string>, default:cMersenneTwister; per-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-run setting
    Setting a nonzero value will cause the $runnumber variable to get padded
```

```
    with leading zeroes to the given length.

<object-full-path>.scalar-recording=<bool>, default:true; per-object setting
    Whether the matching output scalars should be recorded. Syntax:
    <module-full-path>.<scalar-name>.scalar-recording=true/false. Example:
    **.queue.packetsDropped.scalar-recording=true

scheduler-class=<string>, default:cSequentialScheduler; global 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.

seed-%-lcg32=<int>; per-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-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-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-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= key), and ${runnumber}.

sim-time-limit=<double>, unit="s"; per-run setting
    Stops the simulation when simulation time reaches the given limit. The
    default is no limit.

simtime-scale=<int>, default:-12; global setting
    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}-${runnumber}.sna;
per-run setting
    Name of the snapshot file.

snapshotmanager-class=<string>, default:cFileSnapshotManager; global 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
    cSnapshotManager interface.

tkenv-default-config=<string>; global setting
    Specifies which config Tkenv should set up automatically on startup. The
    default is to ask the user.

tkenv-default-run=<int>, default:0; global setting
    Specifies which run (of the default config, see tkenv-default-config) Tkenv
    should set up automatically on startup. The default is to ask the user.

tkenv-extra-stack=<double>, unit="B", default:48KB; global setting
    Specifies the extra amount of stack that is reserved for each activity()
    simple module when the simulation is run under Tkenv.

tkenv-image-path=<path>; global setting
    Specifies the path for loading module icons.

tkenv-plugin-path=<path>; global setting
    Specifies the search path for Tkenv plugins. Tkenv plugins are .tcl files
    that get evaluated on startup.
```

```
total-stack=<double>, unit="B"; global setting
    Specifies the maximum memory for activity() simple module stacks. You need
    to increase this value if you get a ``Cannot allocate coroutine stack''
    error.

<object-full-path>.type-name=<string>; per-object setting
    Specifies type for submodules and channels declared with 'like <>'.

user-interface=<string>; global setting
    Selects the user interface to be started. Possible values are Cmdenv and
    Tkenv. 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.

<object-full-path>.vector-max-buffered-values=<int>; per-object setting
    For output vectors: 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)

<object-full-path>.vector-record-eventnumbers=<bool>, default:true; per-object
setting
    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.

<object-full-path>.vector-recording=<bool>, default:true; per-object setting
    Whether data written into an output vector should be recorded.

<object-full-path>.vector-recording-interval=<custom>; per-object setting
    Recording interval(s) for an output vector. Syntax: [<from>]..[<to>],...
    That is, both start and end of an interval are optional, and intervals are
    separated by comma. Example: ..100, 200..400, 900..

warnings=<bool>, default:true; per-run setting
    Enables warnings.
```

23.2 Predefined Configuration 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
${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
${iterationvars2}
    Concatenation of all user-defined iteration variables in name=value form,
    plus ${repetition}
```

---

24 Appendix: Result File Formats

The file format described here applies to *both output vector and output scalar files*. Their formats are consistent, only the types of entries occurring into 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 don't 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
- **param**: 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

24.1 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 2. Version 1 files are produced by OMNeT++ 3.3 or earlier.

24.2 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 runs 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 OMNeT++ include the following:

Generic attribute names:

- **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
- **iterationvars2**: string containing the values of the iteration variables

An example run header:

```
run TokenRing1-0-20080514-18:19:44-3248
attr configname TokenRing1
attr datetime 20080514-18:19:44
attr experiment TokenRing1
attr inifile omnetpp.ini
attr iterationvars ""
attr iterationvars2 $repetition=0
```

```
attr measurement ""
attr network TokenRing
attr processid 3248
attr repetition 0
attr replication #0
attr resultdir results
attr runnumber 0
attr seedset 0
```

24.3 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"
```

24.4 Module Parameters

Contains a module parameter value for the given run. This is needed so that module parameters may be included in the analysis (e.g. to identify the load for a ``throughput vs load'' plot).

It may not be practical to simply store all parameters of all modules in the result file, because there may be too many. We assume that NED files are invariant and don't store parameters defined in them. However, we store parameter assignments that come from omnetpp.ini, in their original wildcard form (i.e. not expanded) to conserve space. Parameter values entered interactively by the user are also stored.

When the original NED files are present, it should thus be possible to reconstruct all parameters for the given simulation.

Syntax:

param *parameterNamePattern value*

Example:

```
param **.gen.sendIaTime  exponential(0.01)
param **.gen.msgLength   10
param **.fifo.bitsPerSec 1000
```

24.5 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. OMNeT++ uses the following vector attributes:

- **unit**: measurement unit, e.g. `s` for seconds

24.6 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. OMNeT++ uses the following vector attributes:

- **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

24.7 Vector Data

Adds a value to an output vector. This is the same as in older output vector files.

Syntax:

\itshape 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 sizes roughly 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.

24.8 Index Header

The first line of the index file stores the size and modification date of the vector file. If the attributes of vector file differs from the information stored in the index file, then the IDE automatically rebuilds the index file.

Syntax:

file *filesize modificationDate*

24.9 Index Data

Stores the location and statistics of blocks in the vector file.

Syntax:

\itshape 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

24.10 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.

OMNeT++ uses the following attributes:

- **unit**: measurement unit, e.g. `s` for seconds
- **isDiscrete**: boolean (`0` or `1`), meaning whether observations are integers

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 isDiscrete  1
attr unit  packets
bin     -INF    0
bin     -0.5    0
bin     0.5     0
bin     1.5     2254
bin     2.5     2047
bin     3.5     1586
bin     4.5     1428
bin     5.5     1101
bin     6.5     952
bin     7.5     785
```

```
...
bin     51.5    2
```

24.11 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
- **sum**: sum of the observations
- **sqrsum**: sum of the squared observations
- **min**: minimum of the observations
- **max**: maximum of the observations

For weighted statistics, additionally the following fields may be recorded:

- **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

24.12 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 lower bound of the given histogram cell. **Bin** lines are in increasing *binLowerBound* order.

The *value* column of **bin** lines represent observation count in the given cell: *value k* is the number of observations greater or equal than *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
```

---

25 Appendix: 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 gets recorded into the file,

> [With certain granularity of course, and subject to filters that were active during simulation]

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 line-oriented text file. Blank lines and lines beginning with "#" (comments) will get 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 hypen, and continue with the actual text.

```
<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>
```

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

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
```

A correct eventlog also fulfills the following requirements:

- simulation events are in increasing event number and simulation time order

The various entry types and their supported attributes are as follows:

```
SB SimulationBeginEntry // recorded at the first event
{
   v int version       // simulator version, e.g. 0x401 (=1025) is release 4.1
   rid string runId    // identifies the simulation run
}

SE SimulationEndEntry // optional last line of an event log file
```

```
{
}

BU BubbleEntry // display a bubble message
{
   id int moduleId // id of the module which printed the bubble message
   txt string text // displayed message text
}

MB ModuleMethodBeginEntry // beginning of a call to another module
{
   sm int fromModuleId // id of the caller module
   tm int toModuleId   // id of the module being called
   m string method     // C++ method name
}

ME ModuleMethodEndEntry // end of a call to another module
{
}

MC ModuleCreatedEntry // creating a module
{
   id int moduleId              // id of the new module
   c string moduleClassName     // C++ class name of the module
   t string nedTypeName         // fully qualified NED type name
   pid int parentModuleId  -1   // id of the parent module
   n string fullName            // full dotted hierarchic module name
   cm bool compoundModule false // simple or compound module
}

MD ModuleDeletedEntry // deleting a module
{
   id int moduleId // id of the module being deleted
}

MR ModuleReparentedEntry // reparenting a module
{
   id int moduleId         // id of the module being reparented
   p int newParentModuleId // id of the new parent module
}

GC GateCreatedEntry     // gate created
{
   m int moduleId       // module in which the gate was create
   g int gateId         // id of the new gate
   n string name        // gate name
   i int index -1       // gate index if vector, -1 otherwise
   o bool isOutput      // input or output gate
}

GD GateDeletedEntry     // gate deleted
{
   m int moduleId       // module in which the gate was created
   g int gateId         // id of the deleted gate
}

CC ConnectionCreatedEntry // creating a connection
{
   sm int sourceModuleId  // id of the source module identifying the connection
   sg int sourceGateId    // id of the gate at the source module identifying the
connection
   dm int destModuleId    // id of the destination module
   dg int destGateId      // id of the gate at the destination module
}

CD ConnectionDeletedEntry // deleting a connection
{
   sm int sourceModuleId // id of the source module identifying the connection
   sg int sourceGateId   // id of the gate at the source module identifying the
```

```
connection
}

CS ConnectionDisplayStringChangedEntry // a connection display string change
{
   sm int sourceModuleId  // id of the source module identifying the connection
   sg int sourceGateId    // id of the gate at the source module identifying the
connection
   d string displayString // the new display string
}

MS ModuleDisplayStringChangedEntry // a module display string change
{
   id int moduleId        // id of the module
   d string displayString // the new display string
}

E EventEntry // an event that is processing of a message
{
   # long eventNumber          // unique event number
   t simtime_t simulationTime  // simulation time when the event occurred
   m int moduleId              // id of the processing module
   ce long causeEventNumber -1 // event number from which the message being
processed was sent or -1 if the message was sent from initialize
   msg long messageId          // life time unique id of the message being
processed
}

CE CancelEventEntry // canceling an event caused by self message
{
   id long messageId             // id of the message being removed from the FES
   pe long previousEventNumber -1 // event number from which the message being
cancelled was sent or -1 if the message was sent from initialize
}

BS BeginSendEntry // beginning to send a message
{
   id long messageId                       // life time unique id of the message
being sent
   tid long messageTreeId                  // id of the message inherited by dup
   eid long messageEncapsulationId   -1    // id of the message inherited by
encapsulation
   etid long messageEncapsulationTreeId -1 // id of the message inherited by both
dup and encapsulation
   c string messageClassName               // C++ class name of the message
   n string messageFullName                // message name
   pe long previousEventNumber -1          // event number from which the message
being sent was processed or -1 if the message has not yet been processed before
   k short messageKind     0               // message kind
   p short messagePriority 0               // message priority
   l int64 messageLength   0               // message length in bits
   er bool hasBitError   false             // true indicates the message has bit
errors
   d string detail       NULL              // detailed information of message
content when recording message data is turned on
}

ES EndSendEntry // prediction of the arrival of a message
{
   t simtime_t arrivalTime // when the message will arrive to its destination
module
}

SD SendDirectEntry // sending a message directly to a destination gate
{
   sm int senderModuleId            // id of the source module from which the
message is being sent
   dm int destModuleId              // id of the destination module to which the
message is being sent
```

```
   dg int destGateId                // id of the gate at the destination module
to which the message is being sent
   pd simtime_t propagationDelay  0 // propagation delay that is while the
message is propagated through the connection
   td simtime_t transmissionDelay 0 // transmission delay that is while the whole
message is sent from the source gate
}

SH SendHopEntry // sending a message through a connection identified by its
source module and gate id
{
   sm int senderModuleId            // id of the source module from which the
message is being sent
   sg int senderGateId              // id of the gate at the source module from
which the message is being sent
   pd simtime_t propagationDelay  0 // propagation delay that is while the
message is propagated through the connection
   td simtime_t transmissionDelay 0 // transmission delay that is while the whole
message is sent from the source gate
}

DM DeleteMessageEntry // deleting a message
{
   id int messageId               // id of the message being deleted
   pe long previousEventNumber -1 // event number from which the message being
deleted was sent or -1 if the message was sent from initialize
}
```

Document converted from LaTeX by Itoh from Russell W. Quong (quong@best.com)