OMNeT++

Discrete Event Simulation System

Version 2.3

User Manual

by András Varga

Last updated: June 15, 2003

Document History

Date	Author	Change
2003/06	AV	OMNeT++ 2.3 released
2003/04-06	AV	"Design of OMNeT++" chapter revised, extended, and renamed to "Customization and Embedding". Added "Interpreting Cmdenv output" section to the "Running the Simulation" chapter. Added section about Akaroa in "Running the Simulation" chapter. Expanded section about writing shell scripts to control the simulation. Added background info about RNGs and warning about old RNG in "Class Library" chapter; revised/extended "Deriving new classes" section in same chapter. Bibliography converted to Bibtex, expanded and cleaned up; citations added to text. "Parallel Simulation" chapter: contents removed until new PDES implementation gets released. Revised and reorganized NED chapter. Section about message sending/receiving and other simple module related functions moved to chapter "Simple Modules"; cMessage treatment from "Simulation Library" merged with message subclassing chapter into new chapter "Messages". Deprecated cPacket. Removed sections "Simulation techniques" and "Coding conventions", and their useful fragments were incorporated elsewhere. Added/created sections about message transmission modeling, and using global variables. Added sections explaining how to implement broadcasts and retransmissions. Revised section about dynamic module creation. Deprecated putaside-queue, receiveNew(), receiveOn(). Added section "Object ownership management"; removed section on "Using shared objects".
2003/03	AV	OMNeT++ 2.3b2 released
2003/02	AV	OMNeT++ 2.3b1 released
2003/01	AV	Added chapter about message subclassing; revised chapter about running the simulation and incorporated new Cmdenv options; added new distributions and clarified many details in NED expr. handling section
Summer 2002	Ulrich Kaage	Converted from Word to LaTeX
2002/03/18	AV	Documented new ini file options about Envir plugins
2002/01/24	AV	Refinements on the Parsec chapter
2001/10/23	AV	Updated to reflect changes since 2.1 release (see include/ChangeLog)

Contents

Chapter 1

Introduction

1.1 What is OMNeT++?

OMNeT++ is an object-oriented modular discrete event simulator. The name itself stands for Objective Modular Network Testbed in C++. OMNeT++ has its distant roots in OMNeT, a simulator written in Object Pascal by dr. György Pongor.

The simulator can be used for:

- traffic modeling of telecommunication networks
- protocol modeling
- modeling queueing networks
- modeling multiprocessors and other distributed hardware systems
- validating hardware architectures
- evaluating performance aspects of complex software systems
- . . . modeling any other system where the discrete event approach is suitable.

An OMNeT++ model consists of hierarchically nested modules. The depth of module nesting is not limited, which allows the user to reflect the logical structure of the actual system in the model structure. Modules communicate with message passing. Messages can contain arbitrarily complex data structures. Modules can send messages either directly to their destination or along a predefined path, through gates and connections.

Modules can have parameters which are used for three main purposes: to customize module behaviour; to create flexible model topologies (where parameters can specify the number of modules, connection structure etc); and for module communication, as shared variables.

Modules at the lowest level of the module hierarchy are to be provided by the user, and they contain the algorithms in the model. During simulation execution, simple modules appear to run in parallel, since they are implemented as coroutines (sometimes termed lightweight processes). To write simple modules, the user does not need to learn a new programming language, but he/she is assumed to have some knowledge of C++ programming.

OMNeT++ simulations can feature different user interfaces for different purposes: debugging, demonstration and batch execution. Advanced user interfaces make the inside of

the model visible to the user, allow him/her to start/stop simulation execution and to intervene by changing variables/objects inside the model. This is very important in the development/debugging phase of the simulation project. User interfaces also facilitate demonstration of how a model works.

The simulator as well as user interfaces and tools are portable: they are known to work on Windows and on several Unix flavours, using various C++ compilers.

OMNeT++ also supports parallel simulation (as of OMNeT++ 2.3, currently this feature is currently being redesigned.)

1.2 Where is OMNeT++ in the world of simulation tools?

There are numerous network simulation tools on the market today, both commercial and non-commercial. In this section I will try to give an overview by picking some of the most important or most representative ones in both categories and comparing them to OMNeT++: PARSEC, SMURPH, NS, Ptolemy, NetSim++, C++SIM, CLASS as non-commercial, and OPNET, COMNET III as commercial tools. (The OMNeT++ Home Page contains a list of Web sites with collections of references to network simulation tools where the reader can get a more complete list.) In the commercial category, OPNET is widely held to be the state of the art in network simulation. OMNeT++ is targeted at roughly the same segment of network simulation as OPNET.

Seven issues are examined to get an overview about the network simulation tools:

Detail Level. *Does the simulation tool have the necessary power to express details in the model?* In other words, can the user implement arbitrary new building blocks like in OMNeT++ or he is confined to the predefined blocks implemented by the supplier? Some tools like COMNET III are not programmable by the user to this extent therefore they cannot be compared to OMNeT++. Specialized network simulation tools like NS (for IP) and CLASS (for ATM) also rather fall into this category.

Available Models. *What protocol models are readily available for the simulation tool?* On this point, non-commercial simulation tools cannot compete with some commercial ones (especially OPNET) which have a large selection of ready-made protocol models. OMNeT++ is no exception.

Defining Network Topology. *How does the simulation tool support defining the network topology?* Is it possible to create some form of hierarchy (nesting) or only "flat" topologies are supported? Network simulation tools naturally share the property that a model (network) consists of "nodes" (blocks, entities, modules, etc.) connected by "links" (channels, connections, etc.). Many commercial simulators have graphical editors to define the network; however, this is only a good solution if there is an alternative form of topology description (e.g. text file) which allows one to generate the topology by program. OPNET follows a unique way: the network topology is stored in a proprietary binary file format which can be generated (and read) by the graphical editor and C programs linked against a special library. On the other hand, most non-commercial simulation tools do not provide explicit support for topology description: one must program a "driver entity" which will boot the model by creating the necessary nodes and interconnecting them (PARSEC, SMURPH, NS). Finally, a large part of the tools that do support explicit topology description supports only flat topologies (CLASS). OMNeT++ probably uses the most flexible method: it has a human-readable textual topology description format (the NED language) which is easy to create with any text-processing tool (`perl`, `awk`, etc.), and the same format is used by the graphical editor. It is also possible to create a "driver entity" to build a network at run-time by program. OMNeT++ also supports submodule nesting.

Programming Model. *What is the programming model supported by the simulation en-*

vironment? Network simulators typically use either thread/coroutine-based programming (such as `activity()` in OMNeT++), or FSMs built upon a `handleMessage()`-like function. For example, OPNET, SMURPH and NetSim++ use FSMs (with underlying handleMessage()), PARSEC and C++SIM use threads. OMNeT++ supports both programming models; the author does not know of another simulation tool that does so.

Debugging and Tracing Support. *What debugging or tracing facilities does the simulation tool offer?* Simulation programs are infamous for long debugging periods. C++-based simulation tools rarely offer much more than `printf()`-style debugging; often the simulation kernel is also capable of dumping selected debug information on the standard output. Animation is also often supported, either off-line (record&playback) or in some client-server architecture, where the simulation program is the "server" and it can be viewed using the "client". Off-line animation naturally lacks interactivity and is therefore little use in debugging. The client-server solution typically has limited power because the simulation and the viewer run as independent operating system processes, and the viewer has limited access to the simulation program's internals and/or it does not have enough control over the course of simulation execution. OPNET has a very good support for command-line debugging and provides both off-line and client-server style animation. NetSim++ and Ptolemy use the client-server method of animation. OMNeT++ goes a different way by linking the GUI library with the debugging/tracing capability into the simulation executable. This architecture enables the GUI to be very powerful: every user-created object is visible (and modifiable) in the GUI via inspector windows and the user has tight control over the execution. To the author's best knowledge, the tracing feature OMNeT++ provides is unique among the C++-based simulation tools.

Performance. *What performance can be expected from the simulation?* Simulation programs typically run for several hours. Probably the most important factor is the programming language; almost all network simulation tools are C/C++-based. Performance is a particularly interesting issue with OMNeT++ since the GUI debugging/tracing support involves some extra overhead in the simulation library. However, in a reported case, an OMNeT++ simulation was only 1.3 slower than its counterpart implemented in plain C (i.e. one containing very little administration overhead), which is a very good showing. A similar result was reported in a performance comparison with a PARSEC simulation.

Source Availability. *Is the simulation library available in source?* This is a trivial question but it immediately becomes important if one wants to examine or teach the internal workings of a simulation kernel, or one runs into trouble because some function in the simulation library has a bug and/or it is not documented well enough. In general it can be said that non-commercial tools (like OMNeT++) are open-source and commercial ones are not. This is also true for OPNET: the source for simulation kernel is not available (although the ready-made protocol models come with sources).

In conclusion, it can be said that OMNeT++ has enough features to make it a good alternative to most network simulation tools, and it has a strong potential to become one of the most widely used network simulation packages in academic and research environments. The most serious shortcoming is the lack of available protocol models, but since this is mostly due to the fact that it is a relatively new simulation tool, with the help of the OMNeT++ user community the situation is likely to become much better in the future.

1.3 Organization of this manual

The manual is organized around the following topics:

- The Chapters 1, 2 and 3 contain introductory material: some overview and an example simulation.

- The second group of Chapters, 4, 5 and 7 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 following chapters, 8, 9 and 10 deal with practical issues like building and running simulations and analyzing results, and present the tools OMNeT++ has to support these tasks.
- Chapter 11 is devoted to the support for distributed execution.
- Finally, Chapter 12 explains the architecture and the internals of OMNeT++. This chapter will be useful to those who want to extend the capabilities of the simulator or want to embed it into a larger application.
- Appendice A provides a reference of the NED language.

1.4 History

The development of OMNeT++ started as a semester's programming assignment at the Technical University of Budapest (BME) in 1992. The assignment ("creation of an object-oriented discrete event simulation system in C++") was handed out by Prof. Dr György Pongor, and two students signed up: Ákos Kun and András Varga. The basis for the design was Mr. Pongor's existing simulation software written in Pascal, named OMNeT.

We started developing the code in Borland C++ 3.1. The idea of multiple runtime environments, a significant addition to the original OMNeT design, was there from the very beginning. We used Turbo Vision (Borland's then successful character-based GUI) for the first 'graphical' user interface.

In 1992, we submitted a paper about OMNeT++ to the student's annual university conference (named "TDK") and won first prize in the "Software" section. Later we also won 1st prize in the national "TDK". Then the idea came to port OMNeT++ to Unix (first for AIX on an RS/6000 with only 16MB RAM, later Linux), until all development was done in Linux and BC3.1 could no longer be supported.

Well, here's a brief list of events since then – maybe one time I'll make up my mind to enhance them to a whole story...

1994: XEnv (a GUI in pure MOTIF, superceded by Tkenv by now) was written as diploma work

1994: used OPNET for several simulation projects. OPNET features (and flaws) gave lots of ideas how to continue with OMNeT++.

1995: initial version of nedc was written by a group of exchange students from Delft

1996: initial version of PVM support was programmed by Zoltan Vass as diploma work

1997: started working on Tkenv

1997 Dec: added GNED

1997 Sept: web site set up, first public release

1997 Feb-1998 Sept: simulation projects for a small company in Hungary. We used a version of OMNeT++.

1998 March: added Plove

1998 June: animation implemented in Tkenv

1998 Sept-1999 May: work at MeTechnology (later Brokat) in Leipzig

2000 Jan: MSVC porting

2000 Sept: contributed model repository set up

2000: IP-suite created in Karlsruhe

2001 June: the CVS is hosted in Karlsruhe

...

1.5 Authors

OMNeT++ has been developed mostly by András Varga at the Technical University of Budapest, Department of Telecommunications (BME-HIT).

András Varga BME-HIT, andras@whale.hit.bme.hu

Since leaving the university in 1998, I've been doing the development in my free time.

Several people have worked for shorter periods (1..3 months) on different topics within OMNeT++. Credit for organizing this goes to Dr. György Pongor (BME-HIT, pongor@hit.bme.hu), my advisor at the University. Here is a more-or-less complete list of people:

Old NED compiler, 1992-93:
Ákos Kun BME

JAR compiler (now called NEDC), sample simulations; summer 1995:
Jan Heijmans TU Delft
Alex Paalvast TU Delft
Robert van der Leij TU Delft

New feaures, testing, new examples; fall 1995:
Maurits André TU Delft, M.J.A.Andre@twi.tudelft.nl
George van Montfort TU Delft, G.P.R.vanMontfort@twi.tudelft.nl
Gerard van de Weerd TU Delft, G.vandeweerd@twi.tudelft.nl

JAR (NEDC) support for distributed execution:
Gábor Lencse BME-HIT, lencse@hit.bme.hu

PVM support (as final project), spring 1996:
Zoltán Vass BME-HIT

P^2, k-split algorithms and more, from fall 1996:
Babak Fakhamzadeh TU Delft

We have to mention Dr. Leon Rothkranz from the Technical University of Delft whose work made it possible for the Delft students to come to Budapest in 1995.

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 file contains acknowledgements to those I can remember.

Since the summer of 2001, the OMNeT++ sources are kept in the CVS server at the University of Karlsruhe. Credit for setting up and maintaining the CVS server goes to Ulrich Kaage.

The starting point of this manual was the 1995 report of Jan Heijmans, Alex Paalvast and Robert van der Leij.

Chapter 2

Overview

2.1 Modeling concepts

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 with messages. 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. 2.1). 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.

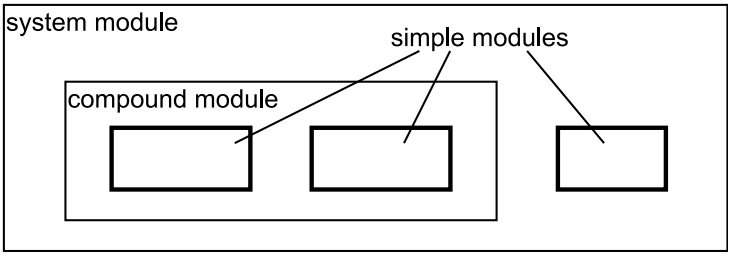

Figure 2.1: Simple and compound modules

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. 2.2).

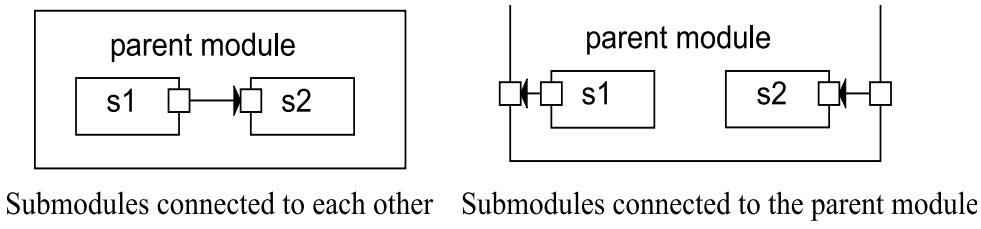

Figure 2.2: Connections

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 their 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 for 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 speficifies 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, the data rate modeling feature of OMNeT++ cannot be used.

2.1.5 Parameters

Modules can have parameters. Parameters are used for three purposes:

1. to parameterize module topology
2. to customize simple module behaviour
3. for module communication, as shared variables

Parameters can take string, numeric or pointer values; 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.

Compound modules can pass parameters or expressions of parameters to their submodules. Parameter passing can be done by value or by reference.

During simulation execution, if a module changes the value of a parameter taken by reference, the changed value propagates to other modules. This effect can be used to tune the model or as a second means of module communication.

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 4.

2.2 Programming the algorithms

The simple modules of a model contain the algorithms as C++ functions. The full flexibility and power of the programming language can be used, supported by the OMNeT++ simulation class library.

OMNeT++ supports a process-style description method for describing activities. During simulation execution, simple module functions appear to run in parallel, because they are implemented as coroutines (also termed lightweight processes). Coroutines were chosen because they allow an intuitive description of the algorithm and they can also serve as a good basis for implementing other description methods like state-transition diagrams or Petri nets.

OMNeT++ has a consistent object-oriented design. One can freely use OOP concepts (inheritance, polymorphism etc) to extend the functionality of the simulator.

Elements of the simulation (messages, modules, queues etc.) are represented as objects. These 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 objects are designed so that they can efficiently work together, creating a powerful framework for simulation programming.

2.2.1 Creating simple modules

Each simple module type is implemented with a C++ class. Simple module classes are derived from a simple module base class, by redefining the virtual function that contains the algorithm. The user can add other member functions to the class to split up a complex algorithm; he can also add data members to the class.

It is also possible to derive new simple module classes from existing ones. For example, if one wants to experiment with retransmission timeout schemes in a transport protocol, he can implement the protocol in one class, create a virtual function for the retransmission algorithm and then derive a family of classes that implement concrete schemes. This concept is further supported by the fact that in the network description, actual module types can be parameters.

2.2.2 Object mechanisms

The use of smart container classes allows the user to build *aggregate data structures*. For example, one can add any number of objects to a message object as parameters. Since the added objects can contain further objects, complex data structures can be built.

There is an efficient *ownership* mechanism built in. The user can specify an owner for each object; then, the owner object will have the responsibility of destroying that object. Most

of the time, the ownership mechanism works transparently; ownership only needs to be explicitly managed when the user wants to do something non-typical.

The *foreach* mechanism allows one to enumerate the objects inside a container object in a uniform way and do some operation on them. This feature which makes it possible to handle many objects together. (The *foreach* feature is extensively used by the user interfaces with debugging capability and the snapshot mechanism; see later.)

2.2.3 Derive new classes

It most cases, the functionality offered by the OMNeT++ classes is enough for the user. But if it is needed, one can derive new classes from the existing ones or create entirely new classes. For flexibility, several member functions are declared virtual. When the user creates new classes, certain rules need to be kept so that the object can fully work together with other objects.

2.2.4 Self-describing objects to ease debugging

The class library is designed so that objects can give textual information about themselves. This makes it possible to peek into a running simulation program: through an appropriate user interface, one can examine (and modify) the internal data structures of a running simulation. This feature helps the user to get some insight what is happening inside the model and get hands-on experience.

A unique feature called *snapshot* allows the user to dump the contents of the simulation model or a part of it into a text file. The file will contain textual reports about every object; this can be of invaluable help at times of debugging. Ordinary variables can also be made to appear in the snapshot file. Snapshot creations can be scheduled from within the simulation program or done from the user interface.

2.3 Using OMNeT++

2.3.1 Building and running simulations

This section gives some idea how to work with OMNeT++ in practice: issues like model files, compiling and running simulations are discussed.

An OMNeT++ model consists of the following parts:

- NED language topology description(s) which describe the module structure with parameters, gates etc. They are files with .ned suffix. NED files can be written with any text editor or using the GNED graphical editor.
- 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 and put together to form a library (a file with .a or .lib extension)
- User interfaces. OMNeT++ user interfaces are used with simulation execution, to facilitate debugging, demonstration, or batch execution of simulations. There are several

user interfaces, written in C++, compiled and put together into libraries (`.a` or `.lib` files).

Simulation programs are built from the above components. First, the NED files are compiled into C++ source code, using the NEDC compiler which is part of OMNeT++. Then all C++ sources are compiled and linked with the simulation kernel and a user interface to form a simulation executable.

Running the simulation and analyzing the results

The simulation executable is a standalone program, thus it can be run on other machines without OMNeT++ or the model files being present. When the program is started, it reads in a configuration file (usually called `omnetpp.ini`); it contains settings that control how the simulation is run, 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++ provides a GUI tool named Plove to view and plot the contents of output vector files. But it is not expected that someone will process the result files using OMNeT++ alone: output files are text files in a format which (maybe after some preprocessing using `sed`, `awk` or `perl`) can be read into math packages like Matlab or its free equivalent Octave, or imported into spreadsheets like Excel. All these external programs have rich functionality for statistical analysis and visualization, and OMNeT++ does not try to duplicate their efforts. This manual briefly describes some data plotting programs and how to use them with OMNeT++.

User interfaces

The primary purpose of user interfaces is to make the inside of the model visible to the user, to start/stop simulation execution, and possibly allow the user 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' about the model's behaviour. A nice graphical user interface can also be used to demonstrate how the model works internally.

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 separately from the place of their actual usage. This means that the user can 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 he/she wants to 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 what in the directories

To help you navigate among files in the OMNeT++ distribution, here's a list what you can find in the different directories.

The `omnetpp` directory contains the following subdirectories.

The simulation system itself:

```
omnetpp/          OMNeT++ root directory
  bin/            OMNeT++ executables (GNED, nedc, etc.)
  include/        header files for simulation models
  lib/            library files
  bitmaps/        icons that can be used in network graphics
  doc/            manual (PDF), readme, license, etc.
    html/         manual in HTML
     api/         API reference in HTML
     nedxml-api/  API reference for the NEDXML library
  src/            OMNeT++ sources
    nedc/         NED compiler
    sim/          simulation kernel
      std/        files for non-distributed execution
      pvm/        files for distributed execution over PVM
      mpi/        files for distributed execution using MPI
    envir/        common code for user interfaces
    cmdenv/       command-line user interface
    tkenv/        Tcl/Tk-based user interface
    gned/         graphical NED editor
    plove/        output vector analyzer and plotting tool
    nedxml/       NEDXML library (experimental)
    utils/        makefile-autocreator etc
  test/           regression test suite
    distrib/      regression test suite for built-in distributions
```

There is a tutorial, contributed by Nick van Foreest

```
tutorial/     the tutorial document
  queues/     sample simulation that supports the tutorial
  doc_src/    the Latex sources for the tutorial doc
```

Sample simulations are within the `samples` directory. Each of the sample directories contain a network description (.ned file) and corresponding simple module code (`.h`, `.cc` files). Makefiles are included.

```
samples/      directories for sample simulations
  nim/        a simple two-player game
  hcube/      hypercube network with deflection routing
  token/      simple model of a Token Ring LAN
  fddi/       an accurate FDDI MAC simulation
  hist/       demo of the histogram classes
  dyna/       model of a client-server network
  dyna2/      a version of dyna, using message subclassing
  fifo1/      single-server queue
  fifo2/      another version of fifo, with handleMessage() and FSM
  topo/       NED demo, shows how to create various topologies
  demo/       several sim. models in a single executable
```

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
```

You may also find additional directories like `msvc/`, which contain integration components for Microsoft Visual C++, etc.

Chapter 3

An Example: The Nim Game

This chapter contains a full example program that can give you some basic idea of using the simulator. An enhanced version of the Nim example can be found among the sample programs.

Nim is an ancient game with two players and a bunch of sticks. The players take turns, removing 1, 2, 3 or 4 sticks from the heap of sticks at each turn. The one who takes the last stick is the loser.

Of course, building a model of the Nim game is not much of a simulation project, but it nicely demonstrates the modeling approach used by OMNeT++.

Describing the model consists of two phases:

- topology description
- defining the operation of components

3.1 Topology

The game can be modelled in OMNeT++ as a network with three modules: the "game" (a manager module) and two players. The modules will communicate by exchanging messages. The "game" module keeps the current number of tokens and organizes the game; in each turn, the player modules receives the number of tokens from the Game module and sends back its move.

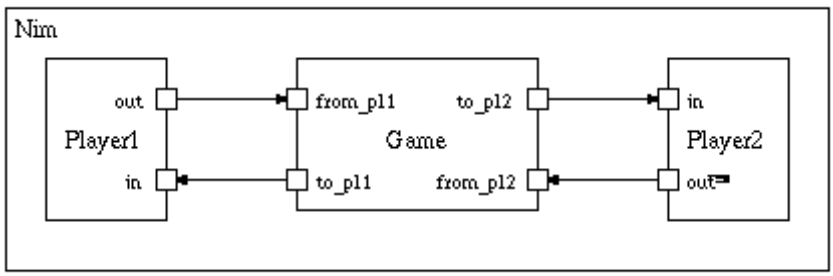

Figure 3.1: Module structure for the Nim game.

`Player1`, `Player2` and `Game` are simple modules (e.g. they have no submodules.) Each

module is an instance of a module type. We'll need a module type to represent the `Game` module; let's call it `Game` too.

We can implement two kinds of players: `SmartPlayer`, which knows the winning algorithm, and `SimplePlayer`, which simply takes a random number of sticks. In our example, `Player1` will be a `SmartPlayer` and `Player2` will be a `SimplePlayer`.

The enclosing module, `Nim` is a compound module (it has submodules). It is also defined as a module type of which one instance is created, the system module.

Modules have input and output gates (the tiny boxes labeled `in`, `out`, `fromPlayer1`, etc. in the figure). An input and an output gate can be connected: connections (or links) are shown as in the figure as arrows. During the simulation, modules communicate by sending messages through the connections.

The user defines the topology of the network in NED files.

We placed the model description in two files; the first file defines the simple module types and the second one the system module.

The first file (NED keywords are typed in boldface):

```
//---------------------------------------------------------
// file: nim_mod.ned
// Simple modules in nim.ned
//---------------------------------------------------------


// Declaration of simple module type Game.

simple Game
    parameters:
         numSticks, // initial number of sticks
         firstMove; // 1=Player1, 2=Player2

    gates:
        in:
             fromPlayer1, // input and output gates
             fromPlayer2; // for connecting to Player1/Player2
        out:
             toPlayer1,
             toPlayer2;
endsimple


// Now the declarations of the two simple module types.
// Any one of the two types can be Player1 or Player2.

// A player that makes random moves
simple SimplePlayer
    gates:
        in: in; // gates for connecting to Game
        out: out;
endsimple

// A player who knows the winning algorithm
simple SmartPlayer
    gates:
```

```
        in: in; // gates for connecting to Game
        out: out;
endsimple
```

The second file:

```
//--------------------------------------------------------------
// file: nim.ned
// Nim compound module + system module
//--------------------------------------------------------------

import "nim_mod";

module Nim
    submodules:
        game: Game
            parameters:
                numSticks = intuniform(21, 31),
                firstMove = intuniform(1, 2);
        player1: SmartPlayer;
        player2: SimplePlayer;
    connections:
        player1.out --> game.fromPlayer1,
        player1.in <-- game.toPlayer1,
        player2.out --> game.fromPlayer2,
        player2.in <-- game.toPlayer2;
endmodule

// system module creation
network
    nim: Nim
endnetwork
```

3.2 Simple modules

The module types `SmartPlayer`, `SimplePlayer` and `Game` are implemented in C++, using the OMNeT++ library classes and functions.

Each simple module type is derived from the C++ class `cSimpleModule`, with its `activity()` member function redefined. The `activity()` functions of all simple modules in the network are executed as coroutines, so they appear as if they were running in parallel. Messages are instances of the class `cMessage`.

We present here the C++ sources of the `SmartPlayer` and `Game` module types.

The `SmartPlayer` first introduces himself by sending its name to the `Game` module. Then it enters an infinite loop; with each iteration, it receives a message from `Game` with the number of sticks left, it calculates its move and sends back a message containing the move.

Here's the source:

```
#include <stdio.h>
#include <string.h>
#include <time.h>
```

```
#include <omnetpp.h>

// derive SmartPlayer from cSimpleModule
class SmartPlayer : public cSimpleModule
{
    Module_Class_Members( SmartPlayer, cSimpleModule, 8192)
    // this is a macro; it expands to constructor definition etc.
    // 8192 is the size for the coroutine stack (in bytes)

virtual void activity();
    // this redefined virtual function holds the algorithm
};

// register the simple module class to OMNeT++
Define_Module( SmartPlayer );

// define operations of SmartPlayer
void SmartPlayer::activity()
{
    int move;

    // initialization phase: send module type to Game module
    // create a message with the name "SmartPlayer" and send it to Game

    cMessage *msg = new cMessage("SmartPlayer");
    send(msg, "out");

    // infinite loop to process moves;
    // simulation will be terminated by Game

    for (;;)
    {
        // messages have several fields; here, we'll use the message
        // kind member to store the number of sticks
        cMessage *msgin = receive();    // receive message from Game
        int numSticks = msgin->kind();  // extract message kind (an int)
                                        // it hold the number of sticks
                                        // still on the stack
        delete msgin;                   // dispose of the message

        move = (numSticks + 4) % 5;     // calculate move
        if (move == 0)                  // we cannot take zero
            move = 1;                   // seems like we going to lose

        ev << "Taking " << move         // some debug output. The ev
           << " out of " << numSticks   // object represents the user
           << " sticks.\n";             // interface of the simulator

        cMessage *msgout = new cMessage;// create empty message
        msgout->setKind( move );        // use message kind as storage
                                        // for move
        send( msgout, "out"); // send the message to Game
    }
}
```

The Game module first waits for a message from both players and extracts the message names that are also the players' names. Then it enters a loop, with the playerToMove variable alternating between 1 and 2. With each iteration, it sends out a message with the current number of sticks to the corresponding player and gets back the number of sticks taken by that player. When the sticks are out, the module announces the winner and ends the simulation.

The source:

```
//--------------------------------------------------------------
// file: game.cc
// (part of NIM - an OMNeT++ demo simulation)
//--------------------------------------------------------------

#include <stdio.h>
#include <string.h>

#include <omnetpp.h>

// derive Game from cSimpleModule
class Game : public cSimpleModule
{
    Module_Class_Members(Game,cSimpleModule,8192)
      // this is a macro; it expands to constructor definition etc.
      // 8192 is the size for the coroutine stack (in bytes)

    virtual void activity();
      // this redefined virtual function holds the algorithm
};

// register the simple module class to OMNeT++
Define_Module( Game );

// operation of Game:
void Game::activity()
{
    // strings to store player names; player[0] is unused
    char player[3][32];

    // read parameter values
    int numSticks = par("numSticks");
    int playerToMove = par("firstMove");

    // waiting for players to tell their names
    for (int i=0; i<2; i++)
    {
        cMessage *msg = receive();
        if (msg->arrivedOn("fromPlayer1"))
            strcpy( player[1], msg->name());
        else
            strcpy( player[2], msg->name());
        delete msg;
    }

    // ev represents the user interface of the simulator
```

```
    ev << "Let the game begin!\n";
    ev << "Player 1: " << player[1] << "   Player 2: " << player[2]
       << "\n\n";

    do
    {
        ev << "Sticks left: " << numSticks << "\n";
        ev << "Player " << playerToMove << " ("
           << player[playerToMove] << ") to move.\n";

        cMessage *msg = new cMessage("", numSticks);
                        // numSticks will be the msg kind

        if (playerToMove == 1)
            send(msg, "toPlayer1");
        else
            send(msg, "toPlayer2");

        msg = receive();
        int sticksTaken = msg->kind();
        delete msg;

        numSticks -= sticksTaken;

        ev << "Player " << playerToMove << " ("
           << player[playerToMove] << ") took "
           << sticksTaken << " stick(s).\n";

        playerToMove = 3 - playerToMove;
    }
    while (numSticks>0);

    ev << "\nPlayer " << playerToMove << " ("
       << player[playerToMove] << ") won!\n";

    endSimulation();
}
```

3.3 Running the simulation

Once the source files are ready, one needs to compile and link them into a simulation executable. One can specify the user interface to be linked.

Before running the simulation, one can put parameter values and all sorts of other settings into an initialization file that will be read when the simulation program starts:

```
#
# file: omnetpp.ini
#

[General]
network = nim
random-seed = 3
```

```
ini-warnings = false

[Cmdenv]
express-mode = no
```

Suppose we link the Nim simulation with the command line user interface. We get the executable `nim` (`nim.exe` under Windows). When we run it, we'll get the following screen output:

```
% ./nim
```

Or:

```
C:\OMNeT++\samples\nim> nim

OMNeT++ Discrete Event Simulation  (C) 1992-2003 Andras Varga
See the license for distribution terms and warranty disclaimer
Setting up Cmdenv (command-line user interface)...

Preparing for Run #1...
Setting up network 'nim'...
Running simulation...

Let the game begin!
Player 1: SmartPlayer Player 2: SimplePlayer

Sticks left: 29
Player 2 (SimplePlayer) to move.
SimplePlayer is taking 2 out of 29 sticks.
Player 2 (SimplePlayer) took 2 stick(s).
Sticks left: 27
Player 1 (SmartPlayer) to move.
SmartPlayer is taking 1 out of 27 sticks.
Player 1 (SmartPlayer) took 1 stick(s).
Sticks left: 26
[...]
Sticks left: 5
Player 1 (SmartPlayer) to move.
SmartPlayer is taking 4 out of 5 sticks.
Player 1 (SmartPlayer) took 4 stick(s).
Sticks left: 1
Player 2 (SimplePlayer) to move.
SimplePlayer is taking 1 out of 1 sticks.
Player 2 (SimplePlayer) took 1 stick(s).

Player 1 (SmartPlayer) won!
<!> Module nim.game: Simulation stopped with endSimulation().

End run of OMNeT++
```

3.4 Other examples

An enhanced version of the Nim example can be found among the sample programs. It adds a third, interactive player and derives specific player types from a `Player` abstract class. It also adds the possibility that actual types for `player1` and `player2` can be specified in the ini file or interactively entered by the user at the beginning of the simulation.

Nim does not show very much of how complex algorithms like communication protocols can be implemented in OMNeT++. To have an idea about that, look at the Token Ring example. It is also extensively commented, though you may need to peep into the user manual to fully understand it. The Dyna simulation models a simple client-server network and demonstrates dynamic module creation. The FDDI example is an accurate FDDI MAC simulation which was written on the basis of the ANSI standard.

The following table summarizes the sample simulations:

NAME	TOPIC	DEMONSTRATES
nim	a simple two-player game	module inheritance; module type as parameter
hcube	hypercube network with deflection routing	hypercube topology with dimension as parameter; topology templates; output vectors
token	Token Ring network	ring topology with the number of nodes as parameter; using `cQueue`; `wait()`; output vectors
fifo1	single-server queue	simple module inheritance; decomposing `activity()` into several functions; using simple statistics and output vectors; printing stack usage info to help optimize memory consumption; using `finish()`
fifo2	another fifo implementation	using `handleMessage()`; decomposing `handleMessage()` into several functions; the FSM macros; simple module inheritance; using simple statistics and output vectors; using `finish()`
fddi	FDDI MAC simulation	using statistics classes; and many other features
hist	demo of the histogram classes	collecting observations into statistics objects; saving statistics objects to file and restoring them
dyna	a client-server network	dynamic module creation; using `WATCH()`; star topology with the number of modules as parameters
topo	various topologies	using NED for creating parametrized topologies
demo	tour of OMNeT++ samples	shows how to link several sim. models into one executable

Chapter 4

The NED Language

4.1 NED overview

The topology of a model is specified using the NED language. The NED language supports modular description of a network. This means that a network description consists of a number of component descriptions (channels, simple/compound module types). The channels, simple modules and compound modules of one network description can be reused in another network description. As a consequence, the NED language makes it possible for users to build their own module libraries.

Files containing network descriptions generally have a `.ned` suffix. NED files are not used directly: they are translated into C++ code by the NEDC compiler, then compiled by the C++ compiler and linked into the simulation executable.

The EBNF description of the language can be found in Appendix A.

4.1.1 Components of a NED description

A NED description can contain the following components, in arbitrary number or order:

- import directives
- channel definitions
- simple and compound module definitions
- network definitions

4.1.2 Reserved words

The writer of the network description has to take care that no reserved words are used for names. The reserved words of the NED language are:

```
import include channel endchannel simple endsimple module endmodule
error delay datarate const parameters gates submodules connections
gatesizes on if machines for do endfor network endnetwork nocheck
ref ancestor true false like input numeric string bool char
```

4.1.3 Identifiers

Identifiers are the names of modules, channels, networks, submodules, parameters, gates, channel attributes and functions.

Identifiers must be composed of letters of the English alphabet (a-z, A-Z), numbers (0-9) and the underscore "_". Identifiers may only begin with a letter or the underscore. If you want to begin an identifier with a digit, prefix the name you'd like to have with an underscore, e.g. `_3Com`.

If you have identifiers that are composed of several words, the convention is to capitalize the beginning of every word. Also, it is recommended that you begin the names of modules, channels and networks with a capital letter, and the names of parameters, gates and submodules with a lower-case letter. Underscores are rarely used.

4.1.4 Case sensitivity

The network description and all identifiers in it are case sensitive. For example, `TCP` and `Tcp` are two different names.

4.1.5 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. Comments are ignored by the NED compiler.

It is planned that future OMNeT++ versions will use comments for documentation generation, much like JavaDoc or Doxygen.

4.2 The import directive

The `import` directive is used to import declarations from another network description file. After importing a network description, one can use the components (channels, simple/compound module types) defined in it.

When a file is imported, only the declaration information is used. Also, importing a `.ned` file does not cause that file to be compiled with the NED compiler when the parent file is NEDC compiled, i.e., one must compile and link all network description files – not only the top-level ones.

You can specify the name of the files with or without the `.ned` extension. You can also include a path in the filenames, or better, use the NEDC compiler's `-I <path>` command-line option to name the directories where the imported files reside.

Example:

```
import "ethernet";   // imports ethernet.ned
```

4.3 Channel definitions

A channel definition specifies a connection type of given characteristics. The channel name can be used later in the NED description to create connections with these parameters.

The syntax:

```
channel ChannelName
    //...
endchannel
```

Three attributes can be assigned values in the body of the channel declaration, all of them optional: `delay`, `error` and `datarate`. `delay` is the propagation delay in (simulated) seconds; `error` is the bit error rate that speficifies the probability that a bit is incorrectly transmitted; and `datarate` is the channel bandwidth in bits/second, used for calculating transmission time of a packet. The attributes can appear in any order.

The values must be constants, or expressions that do not contain external references (e.g. names of module parameters). If you assign a random-valued expression (e.g. `truncnormal(0.005,0.001)`), it will be evaluated and a new random number generated for each packet transmission.

Example:

```
channel DialUpConnection
    delay normal (0.004, 0.0018)
    error 0.00001
    datarate 14400
endchannel
```

4.4 Simple module definitions

Simple modules are the basic building blocks for other (compound) modules. Simple module types are identified by names. By convention, module names begin with upper-case letters.

A simple module is defined by declaring its parameters and gates.

Simple modules are declared with the following syntax:

```
simple SimpleModuleName
    parameters:
        //...
    gates:
        //...
endsimple
```

4.4.1 Simple module parameters

Parameters are variables that belong to a module. Simple module parameters can be queried and used by simple module algorithms. For example, a module called `TrafficGen` may have a parameter called `numOfMessages` that determines how many messages it should generate.

Parameters are identified by names. By convention, parameter names begin with lower-case letters.

Parameters are declared by listing their names in the parameters: section of a module description. The parameter type can optionally be specified as `numeric`, `numeric const` (or simply `const`), `bool`, `string`, or `anytype`.

Example:

```
simple TrafficGen
    parameters:
```

```
        interArrivalTime,
        numOfMessages : const,
        address : string;
    gates: //...
endsimple
```

If the parameter type is omitted, `numeric` is assumed. Practically, this means that you only need to explicitly specify the type for string, bool or char-valued parameters.

Note that the actual parameter values are given later, when the module is used as a building block of a compound module type or as a system module.

Const parameters

When you declare a parameter to be `const`, it will be evaluated and replaced by the resulting constant value at the beginning of the simulation. This can be important when the original value was a random number or an expression. One is advised to write out the `const` keyword for each parameter that should be constant.

Beware when using `const` and by-reference parameter passing (`ref` modifier, see later) at the same time. Converting the parameter to constant can affect other modules and cause errors that are difficult to discover.

4.4.2 Simple module gates

Gates are the connection points of modules. The starting and ending points of the connections between modules are gates. OMNeT++ supports simplex (one-directional) connections, so there are two kinds of gates: input and output. Messages are sent through output gates and received through input gates.

Gates are identified with their names. By convention, gate names begin with lower-case letters.

Gate vectors are supported: a gate vector contains a number of single gates.

Gates are declared by listing their names in the `gates:` section of a module description. An empty bracket pair [] denotes a gate vector. Elements of the vector are numbered starting with zero.

Examples:

```
simple DataLink
    parameters: //...
    gates:
        in:  fromPort, fromHigherLayer;
        out: toPort, toHigherLayer;
endsimple

simple RoutingModule
    parameters: //...
    gates:
        in:  output[];
        out: input[];
endsimple
```

The sizes of gate vectors are given later, when the module is used as a building block of a

compound module type. Thus, every instance of the module can have gate vectors of different sizes.

4.5 Compound module definitions

Compound modules are modules composed of one or more submodules. Any module type (simple or compound module) can be used as a submodule. Like simple modules, compound modules can also have gates and parameters, and they can be used wherever simple modules can be used.

It is useful to think about compound modules as "cardboard boxes" that help you organize your simulation model and bring structure into it. No active behaviour is associated with compound modules – they are simply for grouping modules into larger components that can can be used either as a model (see section 4.6) or as a building block for other compound modules.

By convention, module type names (and so compound module type names, too) begin with upper-case letters.

Submodules may use parameters of the compound module. They may be connected with each other and/or with the compound module itself.

A compound module definition looks similar to a simple module definition: it has `gates` and `parameters` sections. There are two additional sections, `submodules` and `connections`.

The syntax for compound modules is the following:

```
module CompoundModule
    parameters:
        //...
    gates:
        //...
    submodules:
        //...
    connections:
        //...
endmodule
```

All sections (`parameters`, `gates`, `submodules`, `connections`) are optional.

4.5.1 Compound module parameters and gates

Parameters and gates for compound modules are declared and work in the same way as with simple modules, described in sections 4.4.1 and 4.4.2.

Typically, compound module parameters are passed to submodules and used for initializing their parameters.

Parameters can also be used in defining the internal structure of the compound module: the number of submodules, gate vector sizes can be define with the help of parameters, and parameters can also be used in defining the connections inside the compound module. As a practical example, you can create a `Router` compound module with a variable number of ports, specified in a `numOfPorts` parameter.

Parameters affecting the internal structure should always be declared `const`, so that accessing them always yields the same value. (Otherwise, if the parameter was assigned a random

value, one could get a different value each time the parameter is accessed during building the internals of the compound module, which is surely not what was meant.)

Example:

```
module Router
    parameters:
        packetsPerSecond : numeric,
        bufferSize : numeric,
        numOfPorts : const;
    gates:
        in: inputPort[];
        out: outputPort[];
    submodules: //...
    connections: //...
endmodule
```

4.5.2 Submodules

Submodules are defined in the `submodules:` section of a compound module declaration. Submodules are identified by names. By convention, submodule names begin with lowercase letters.

Submodules are instances of a module type, either simple or compound – there is no distinction. The module type must be known to the NED compiler, that is, it must have appeared earlier in the same NED file or have been imported from another NED file.

It is possible to define vectors of submodules, and the size of the vector may come from a parameter value.

When defining submodules, you can assign values to their parameters, and if the corresponding module type has gate vectors, you have to specify their sizes.

Example:

```
module CompoundModuleName
    //...
    submodules:
        submodule1: ModuleType1
            parameters:
                //...
            gatesizes:
                //...
        submodule2: ModuleType2
            parameters:
                //...
            gatesizes:
                //...
endmodule
```

Module vectors

It is possible to create an array of submodules (a module vector). This is done with an expression between brackets right behind the module type name. The expression can refer to module parameters. A zero value as module count is also allowed.

Example:

```
module BigCompoundModule
    parameters:
        size: const;
    submodules:
        submod1: Node[3]
            //...
        submod2: Node[size]
            //...
        submod3: Node[2*size+1]
            //...
endmodule
```

4.5.3 Submodule type as parameter

Sometimes it is convenient to make the name of a submodule type a parameter, so that one can easily 'plug in' any module there.

For example, assume the purpose of your simulation study is to compare different routing algorithms. Suppose you programmed the needed routing algorithms as simple modules: `DistVecRoutingNode`, `AntNetRouting1Node`, `AntNetRouting2Node`, etc. You have also created the network topology as a compound module called `RoutingTestNetwork`, which will serve as a testbed for your routing algorithms. Currently, `RoutingTestNetwork` has `DistVecRoutingNode` hardcoded (all submodules are of this type), but you want to be able to switch to other routing algorithms easily.

NED gives you the possibility to add a string-valued parameter, say `routingNodeType` to the `RoutingTestNetwork` compound module. Then you can tell NED that types of the submodules inside `RoutingTestNetwork` are not of any fixed module type, but contained in the `routingNodeType` parameter. That's all – now you are free to assign any of the `"DistVecRoutingNode"`, `"AntNetRouting1Node"` or `"AntNetRouting2Node"` string constants to this parameter (you can do that in NED, in the config file (`omnetpp.ini`), or even enter it interactively), and your network will use the routing algorithm you chose.

If you specify a wrong value, say `"FooBarRoutingNode"` whereas you have no `FooBarRoutingNode` module implemented, you'll get a runtime error at the beginning of the simulation: *module type definition not found*.

Inside the `RoutingTestNetwork` module you assign parameter values and connect the gates of the routing modules. To provide some degree of type safety, NED wants to make sure you didn't misspell parameter or gate names and you used them correctly. To be able to do such checks, NED requires some help from you: you have to name an existing module type (say `RoutingNode`) and promise NED that all modules you're going you specify in the `routingNodeType` parameter will have (at least) the same parameters and gates as the `RoutingNode` module. [1]

All the above is achieved via the `like` keyword. The syntax is the following:

```
module RoutingTestNetwork
    parameters:
        routingNodeType: string; // should hold the name
                                 // of an existing module type
    gates: //...
    submodules:
```

[1] If you like, the above solution somewhat similar to polymorphism in object-oriented languages – `RoutingNode` is like a "base class", `DistVecRoutingNode` and `AntNetRouting1Node` are like "derived classes", and the `routingNodeType` parameter is like a "pointer to a base class" which may be downcast to specific types.

```
        node1: routingNodeType like RoutingNode;
        node2: routingNodeType like RoutingNode;
        //...
    connections nocheck:
        node1.out0 --> node2.in0;
        //...
endmodule
```

The `RoutingNode` module type does not need to be implemented in C++, because no instance of it is created; it is merely used to check the correctness of the NED file.

On the other hand, the actual module types that will be substituted (e.g. `DistVecRoutingNode`, `AntNetRouting1Node`,etc.) do not need to be declared in the NED files.

The `like` phrase lets you create families of modules that serve similar purposes and implement the same interface (they have the same gates and parameters) and to use them interchangeably in NED files.

4.5.4 Assigning values to submodule parameters

If the module type used as submodule has parameters, you can assign values to them in the `parameters` section of the submodule declaration. As a value you can use a constant (such as `42` or `"www.foo.org"`), various parameters (most commonly, parameters of the compound module), or write an arbitrary expression containing the above.

It is not mandatory to mention and assign all parameters. Unassigned parameters can get their values at runtime: either from the configuration file (`omnetpp.ini`), or if the value isn't there either, the simulator will prompt you to enter it interactively. Indeed, for flexibility reasons it is often very useful not to "hardcode" parameter values in the NED file, but to leave them to `omnetpp.ini` where they can be changed more easily.

Example:

```
module CompoundModule
    parameters:
        param1: numeric,
        param2: numeric,
        useParam1: bool;
    submodules:
        submodule1: Node
            parameters:
                p1 = 10,
                p2 = param1+param2,
                p3 = useParam1==true ? param1 : param2;
        //...
endmodule
```

The expression syntax is very similar to C. Expressions may contain constants (literals) and parameters of the compound module being defined. Parameters can be passed by value or by reference. The latter means that the expression is evaluated at runtime each time its value is accessed (e.g. from simple module code), opening up interesting possibilities for the modeler. You can also refer to parameters of the already defined submodules, with the syntax `submodule.parametername` (or `submodule[index].parametername`).

Expressions are described in detail in section 4.7.

The `input` keyword

When a parameter does not receive a value inside NED files or in the configuration file (`omnetpp.ini`), the user will be prompted to enter its value at the beginning of the simulation. If you plan to make use of interactive prompting, you can specify a prompt text and a default value.

The syntax is the following:

```
<paramname> = input(<default-value>, <prompt>)
<paramname> = input(<default-value>)
<paramname> = input
```

The third version is actually equivalent to simply and quietly leaving out the parameter from the list of assignments, but you can use it to make it explicit that you do not want to assign a value from within the NED file.

Examples:

```
   parameters:
      numProc = input(10, "Number of processors?"),
      processingTime = input(10ms);
```

4.5.5 Defining sizes of submodule gate vectors

The sizes of gate vectors are defined with the `gatesizes` keyword. Gate vector sizes can be given as constants, parameters or expressions.

An example:

```
simple Node
    gates:
        in: inputs[];
        out: outputs[];
endsimple

module CompoundModule
    parameters:
        numPorts: const;
    submodules:
        node1: Node
            gatesizes:
                inputs[2], outputs[2];
        node2: Node
            gatesizes:
                inputs[numPorts], outputs[numPorts];
        //...
endmodule
```

4.5.6 Conditional parameters and gatesizes sections

Multiple `parameters` and `gatesizes` sections can exist in a submodule definition and each of them can be tagged with conditions.

Example:

```
module Tandem
    parameters: count: const;
    submodules:
        node : Node [count]
            parameters:
                position = "middle";
            parameters if index==0:
                position = "beginning";
            parameters if index==count-1:
                position = "end";
            gatesizes:
                in[2], out[2];
            gatesizes if index==0 || index==count-1:
                in[1], in[1];
    connections:
        //...
endmodule
```

If the conditions are not disjoint and a parameter value or a gate size is defined twice, the last definition will take effect, overwriting the former ones. Thus, values intended as defaults should appear in the first sections.

4.5.7 Connections

The compound module definition specifies how the gates of the compound module and its immediate sub-modules are connected.

You can connect two submodules or a submodule with its enclosing compound module. (For completeness, you can also connect two gates of the compound module on the inside, but this is rarely needed). This means that NED does not permit connections that span multiple levels of hiearchy – this restriction enforces compound modules to be self-contained, and thus promotes reusability. Gate directions must also be observed, that is, you cannot connect two output gates or two input gates.

Only one-to-one connections are supported. One-to-many and many-to-one connections can be achieved using simple modules that duplicate messages or merge message flows. The rationale is that wherever such fan-in or fan-out occurs in a model, it is usually associated with some processing anyway that makes it necessary to use simple modules.

A gate can only be connected once: if two connections refer to the same gate, a compilation or runtime error will occur.

By default, NED expects every gate to be connected, resulting in a compilation or runtime error if an unconnected gate is found. This check can be turned off with the `nocheck` modifier, described later in this section.

Connections are specified in the `connections:` section of a compound module definition. It lists the connections, separated by semicolons.

Example:

```
module CompoundModule
    parameters: //...
    gates: //...
    submodules: //...
    connections:
        node1.output --> node2.input;
```

```
        node1.input <-- node2.output;
        //...
endmodule
```

Each connection can be:

- simple (that is, no delay, bit error rate or data rate), can use a named channel, or a channel given with delay, error and data rate values;
- single or multiple (loop) connection;
- conditional or non-conditional.

These connection types are described in the following sections.

Single connections and channels

The source gate can be an output gate of a submodule or an input gate of the compound module, and the destination gate can be an input gate of a submodule or an output gate of the compound module.

If you do not specify a channel, the connection will have no propagation delay, no transmission delay and no bit errors:

```
    sender.outGate --> receiver.inGate;
```

The arrow can point either left-to-right or right-to-left.

You can specify a channel by its name:

```
    sender.outGate --> Dialup14400 --> receiver.inGate;
```

In this case, the NED sources must contain the definition of the channel.

One can also specify the channel parameters directly:

```
    sender.outGate --> error 1e-5 delay 0.001 --> receiver.inGate;
```

Either of the parameters can be omitted and they can be in any order.

Loop connections

If submodule or gate vectors are used, it is possible to create more than one connection with one statement. This is termed a *multiple* or *loop connection*.

A multiple connection is created with the `for` statement:

```
for i=0..4 do
    sender.outGate[i] --> receiver[i].inGate
endfor;
```

The result of the above loop connection can be illustrated as depicted in Fig. 4.1.

One can place several connections in the body of the `for` statement, separated by semicolons.

One can create nested loops by specifying more than one indices in the `for` statement, with the first variable forming the outermost loop.

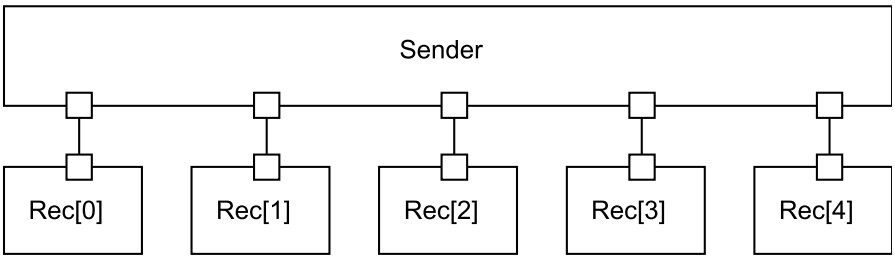

Figure 4.1: Loop connection

```
for i=0..4, j=0..4 do
    //...
endfor;
```

One can also use an index in the lower and upper bound expressions of the subsequent indices:

```
for i=0..3, j=i+1..4 do
    //...
endfor;
```

Conditional connections

Creation of a connection can be made conditional, using the `if` keyword:

```
for i=0..n do
    sender.outGate[i] --> receiver[i].inGate if i%2==0;
endfor;
```

The `if` condition is evaluated for each connection (in the above example, for each *i* value), and the decision is made individually each time whether to create the the connection or not. In the above example we connected every second gate. Conditions may also use random variables, as shown in the next section.

The nocheck modifier

By default, NED requires that all gates be connected. Since this check can be inconvenient at times, it can be turned off using the `nocheck` modifier.

The following example generates a random subgraph of a full graph.

```
module RandomConnections
    parameters: //..
    gates: //..
    submodules: //..
    connections nocheck:
        for i=0..n-1, j=0..n-1 do
            node[i].out[j] --> node[j].in[i] if uniform(0,1)<0.3;
        endfor;
endmodule
```

When using `nocheck`, it is the simple modules' responsibility not to send messages on gates that are not connected.

4.6 Network definitions

Module module declarations (compound and simple module declarations) just define module types. To actually get a simulation model that can be run, you need to write a *network definition*.

A network definition declares a simulation model as an instance of a previously defined module type. You'll typically want to use a compound module type here, although it is also possible to program a model as a self-contained simple module and instantiate it as a "network".

There can be several network definitions in your NED file or NED files. The simulation program that uses those NED files will be able to run any of them; you typically select the desired one in the config file (`omnetpp.ini`).

The syntax of a network definition is similar that of a submodule declaration:

```
network wirelessLAN: WirelessLAN
    parameters:
        numUsers=10,
        httpTraffic=true,
        ftpTraffic=true,
        distanceFromHub=truncnormal(100,60);
endnetwork
```

Here, `WirelessLAN` is the name of previously defined compound module type, which presumably contains further compound modules of types `WirelessHost`, `WirelessHub`, etc.

Naturally, only compound module types without gates can be used in network definitions.

Just as for submodules, you do not need to assign values to all parameters. Unassigned parameters will get their values from the config file (`omnetpp.ini`) or interactively prompted for.

4.7 Expressions

In the NED language there are a number of places where expressions are expected.

Expressions have a C-style syntax. They are built with the usual math operators; they can use parameters taken by value or by reference; call C functions; contain random and input values etc.

When an expression is used for a parameter value, it is evaluated each time the parameter value is accessed (unless the parameter is declared `const`, see 4.4.1). This means that a simple module querying a non-const parameter during simulation may get different values each time (e.g. if the value involves a random variable, or it contains other parameters taken by reference). Other expressions (including `const` parameter values) are evaluated only once.

4.7.1 Constants

Numeric and string constants

Numeric constants are accepted in their usual decimal or scientific notations.

String constants

String constants use double quotes.

Time constants

Anywhere you would put numeric constants (integer or real) to mean time in seconds, you can also specify the time in units like milliseconds, minutes or hours:

```
...
parameters:
    propagationDelay = 560ms, // 0.560s
    connectionTimeout = 6m 30s 500ms, // 390.5s
    recoveryIntvl = 0.5h; // 30 min
```

The following units can be used:

Unit	Meaning	Seconds
ns	nanoseconds	$*10^{-9}$
us	microseconds	$*10^{-6}$
ms	milliseconds	$*10^{-3}$
s	seconds	$*1$
m	minutes	$*60$
h	hours	$*3600$
d	days	$*24*3600$

4.7.2 Referencing parameters

Expressions can use the parameters of the enclosing compound module (the one being defined) and of submodules defined earlier in NED file. The syntax for the latter is `submod.param` or `submod[index].param`.

You can refer to a compound module parameter called `param` in several ways: as `param`, **`ref`** `param`, **`ancestor`** `param`, or **`ref ancestor`** `param`. They all have different semantics.

The first two variations, `param` and **`ref`** `param` lets you access the parameters of the compound module being defined. In the third and fourth versions, the keyword `ancestor` means that the parameter will be searched for upwards, in the module nesting hierarchy. Naturally, this kind of reference can only be resolved at runtime, when the whole network has been built up. The parameter which is found first is used. If no such parameter can be found in any of the enclosing modules, it is a runtime error.

The `ref` and `ref`-less versions differ in how the parameter is taken: by value or by reference. If you take a parameter by reference, then runtime changes to that parameter will be reflected in the assigned parameter: each time a simple module reads the parameter value, the expression is evaluated, and you may get a different value. In contrast, if you take the

parameter by value, then runtime changes do not affect the assigned parameter. [2]

Reference parameters open up interesting possibilities for the modeler. For example, you can define a parameter that at the highest level of the model, and let other modules take it by reference – then if you change the parameter value at runtime (manually or from a simple module), it will affect the whole model. You can use this arrangement to "tune" model parameters at runtime, in search for an optimal setting.

In another setup, reference parameters can be used by to propagate status values to neighbouring modules.

4.7.3 Operators

The set of operators supported in NED is similar to C/C++, with the following differences:

- ^ is used for power-of (and not bitwise XOR as in C)
- # is used for logical XOR (same as != between logical values), and ## is used for bitwise XOR
- the precedence of bitwise operators (&, |, #) have been raised to bind stronger than relational operations. This precedence is usually more convenient than the C/C++ one.

All values are represented as `doubles`. For the bitwise operators, `doubles` are converted to `unsigned long` [3] using the C/C++ builtin conversion (type cast), the operation is performed, then the result is converted back to `double`. Similarly, for the logical operators &&, || and ##, the operands are converted to `bool` using the C/C++ builtin conversion (type cast), the operation is performed, then the result is converted back to `double`. For modulus (%), the operands are converted to `long`.

Here's the complete list of operators, in order of decreasing precendence:

Operator	Meaning
-, !, ~	unary minus, negation, bitwise complement
^	power-of
*, /, %	multiply, divide, modulus
+, -	add, subtract
«, »	bitwise shifting
&, |, #	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"

[2] This also means that if an expression doesn't contain any parameter taken by reference, the NED compiler may have the possibility to evaluate the expression only once, at compile time. If an expression refers to parameters taken by reference, only runtime evaluation can be used.

[3] In case you are worried about `long` values being not accurately represented in `doubles`, this is not the case. IEEE-754 `doubles` have 52 bit mantissas, and integer numbers in that range are represented without rounding errors.

4.7.4 The `sizeof()` and `index` operators

A useful operator is `sizeof()`, which gives the size of a vector gate. The `index` operator gives the index of the current submodule in its module vector.

An example for both:

```
module Compound
    gates: in: fromgens[];
    submodules:
        proc: Processor[ sizeof(fromgens) ];
            parameters: address = 10*(1+index);
    connections:
        for i = 0.. sizeof(fromgens)-1 do
            in[i] --> proc[i].input;
        endfor;
endmodule
```

Here, we create as many processors as there are input gates for this compound module in the network. The address parameters of the processors are 10, 20, 30 etc.

4.7.5 Functions

In NED expressions, you can use the following mathematical functions:

- many of the C language's `<math.h>` library functions: `exp()`, `log()`, `sin()`, `cos()`, `floor()`, `ceil()`, `etc.`
- functions that generate random variables: `uniform`, `exponential`, `normal` and others were already discussed.

It is possible to add new ones, see 4.7.7.

4.7.6 Random values

Expressions may contain random variates from different distributions. This has the effect that unless the parameter was declared `const`, it returns a different value each time it is evaluated.

If the parameter *was* declared `const`, it is only evaluated once at the beginning of the simulation, and subsequent queries on the parameter will always return the same value.

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

If you do not specify the optional `rng` argument, the functions will use random number generator 0.

Examples:

```
intuniform(0,10)/10  // one of: 0, 0.1, 0.2, ..., 0.9, 1
exponential(5)       // exponential with mean=5 (thus parameter=0.2)
2+truncnormal(5,3)   // normal distr with mean 7 truncated to >=2 values
```

The above distributions are implemented with C functions, and you can easily add new ones (see section 4.7.7). Your distributions will be treated in the same way as the built-in ones.

4.7.7 Defining new functions

To use user-defined functions, one has to code the function in C++. The C++ function must take 0, 1, 2, 3, or 4 arguments of type double and return a double. The function must be registered in one of the C++ files with the `Define_Function()` macro.

An example function (the following code must appear in one of the C++ sources):

```
#include <omnetpp.h>
```

```
double average(double a, double b)
{
  return (a+b)/2;
}

Define_Function(average, 2);
```

The number 2 means that the `average()` function has 2 arguments. After this, the `average()` function can be used in NED files:

```
module Compound
    parameter: a,b;
    submodules:
        proc: Processor
            parameters: av = average(a,b);
endmodule
```

If your function takes parameters that are `int` or `long` or some other type which is not `double`, you can create wrapper function that takes all doubles and does the conversion. In this case you have to register the wrapper function with the `Define_Function2()` macro which allows a function to be registered with a name different from the name of the function that implements it. You can do the same if the return value differs from `double`.

```
#include <omnetpp.h>

long factorial(int k)
{
  ...
}

static double _wrap_factorial(double k)
{
  return factorial((int)k);
}

Define_Function2(factorial, _wrap_factorial, 1);
```

4.8 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:` phrases.

The display string format is a semicolon-separated list of tags. Each tag consists of a key (usually one letter), an equal sign and a comma-separated list of parameters, like:

```
  "p=100,100;b=60,10,rect;o=blue,black,2"
```

Parameters may be omitted also at the end and also inside the parameter list, like:

```
  "p=100,100;b=,,rect;o=blue,black"
```

Module/submodule parameters can be included with the `$name` notation:

```
"p=$xpos,$ypos;b=rect,60,10;o=$fillcolor,black,2"
```

Objects that may have display strings are:

- compound modules (as the enclosing module in the drawing),
- submodules
- connections

4.8.1 Submodule display strings

The following table lists the tags used in submodule display strings:

Tag	Meaning
p=*xpos*,*ypos*	Place submodule at (*xpos*,*ypos*) pixel position, with the origin being the top-left corner of the enclosing module. Defaults: an appropriate automatic layout is where submodules do not overlap. If applied to a submodule vector, *ring* or *row* layout is selected automatically.
p=*xpos*,*ypos*,**row**,*deltax*	Used for module vectors. Arranges submodules in a row starting at (*xpos*,*ypos*), keeping *deltax* distances. Defaults: *deltax* is chosen so that submodules do not overlap. **row** may be abbreviated as **r**.
p=*xpos*,*ypos*,**column**,*deltay*	Used for module vectors. Arranges submodules in a column starting at (*xpos*,*ypos*), keeping *deltay* distances. Defaults: *deltay* is chosen so that submodules do not overlap. **column** may be abbreviated as **col** or **c**.
p=*xpos*,*ypos*,**matrix**, *itemsperrow*,*deltax*,*deltay*	Used for module vectors. Arranges submodules in a matrix starting at (*xpos*,*ypos*), at most *itemsperrow* submodules in a row, keeping *deltax* and *deltay* distances. Defaults: *itemsperrow*=5, *deltax,deltay* are chosen so that submodules do not overlap. **matrix** may be abbreviated as **m**.
p=*xpos*,*ypos*,**ring**,*width,height*	Used for module vectors. Arranges submodules in an ellipse, with the top-left corner of its bounding boxes at (*xpos*,*ypos*), with the *width* and *height*. Defaults: *width*=40, *height*=24 **ring** may be abbreviated as **ri**.

p=*xpos*,*ypos*,**exact**,*deltax*,*deltay*	Used for module vectors. Each submodule is placed at *(xpos+deltax*, *ypos+deltay)*. This is useful if *deltax* and *deltay* are parameters (e.g.:*"p=100,100,exact,$x,$y"*) which take different values for each module in the vector. Defaults: *none* **exact** may be abbreviated as **e** or **x**.
b=*width*,*height*,**rect**	Rectangle with the given *height* and *width*. Defaults: *width*=40, *height*=24
b=*width*,*height*,**oval**	Ellipse with the given *height* and *width*. Defaults: *width*=40, *height*=24
o=*fillcolor*,*outlinecolor*,*borderwidth*	Specifies options for the rectangle or oval. Any valid Tk color specification is accepted: English color names or #*rgb*, #*rrggbb* format (where *r*,*g*,*b* are hex digits). Defaults: *fillcolor*=#8080ff (a lightblue), *outlinecolor*=black, *borderwidth*=2
i=*iconname*	Use the named icon. No default. If no icon name is present, *box* is used.

Examples:

```
"p=100,60;i=workstation"
"p=100,60;b=30,30,rect;o=4"
```

4.8.2 Compound module display strings

The tags that can be used in enclosing module display strings are:

Tag	Meaning
p=*xpos*,*ypos*	Place enclosing module at (*xpos*,*ypos*) pixel position, with (0,0) being the top-left corner of the window.
b=*width*,*height*,**rect**	Display enclosing module as a rectangle with the given *height* and *width*. Defaults: *width, height* are chosen automatically
b=*width*,*height*,**oval**	Display enclosing module as an ellipse with the given *height* and *width*. Defaults: *width, height* are chosen automatically
o=*fillcolor*,*outlinecolor*,*borderwidth*	Specifies options for the rectangle or oval. Any valid Tk color specification is accepted: English color names or #*rgb*, #*rrggbb* format (where *r*,*g*,*b* are hex digits). Defaults: *fillcolor*=#8080ff (a lightblue), *outlinecolor*=black, *borderwidth*=2

4.8.3 Connection display strings

Tags that can be used in connection display strings:

Tag	Meaning
m=auto **m=north** **m=west** **m=east** **m=south**	Drawing mode. Specifies the exact placement of the connection arrow. The arguments can be abbreviated as a,e,w,n,s.
m=manual,*srcpx*,*srcpy*,*destpx*,*destpy*	The manual mode takes four parameters that explicitly specify anchoring of the ends of the arrow: *srcpx*, *srcpy*, *destpx*, *destpy*. Each value is a percentage of the width/height of the source/destination module's enclosing rectangle, with the upper-left corner being the origin. Thus, `m=m,50,50,50,50` would connect the centers of the two module rectangles.
o=*color*,*width*	Specifies the appearance of the arrow. Any valid Tk color specification is accepted: English color names or #rgb, #rrggbb specification (where r,g,b are hex digits). Defaults: *color*=black, *width*=2

Examples:

```
"m=a;o=blue,3"
```

4.9 Parameterized compound modules

With the help of conditional parameter and gatesize blocks and conditional connections, one can create complex topologies.

4.9.1 Examples

Example 1: Router

The following example contains a router module with the number of ports taken as parameter. The compound module is built using three module types: Application, RoutingModule, DataLink. We assume that their definition is in a separate NED file which we will import.

```
import "modules";
module Router
    parameters:
        rteProcessingDelay, rteBuffersize,
        numOfPorts: const;
    gates:
        in: inputPorts[];
        out: outputPorts[];
    submodules:
        localUser: Application;
        routing: RoutingModule
```

```
            parameters:
                processingDelay = rteProcessingDelay,
                buffersize = rteBuffersize;
            gatesizes:
                input[numOfPorts+1],
                output[numOfPorts+1];
        portIf: DataLink[numOfPorts]
            parameters:
                retryCount = 5,
                windowSize = 2;
    connections:
        for i=0..numOfPorts-1 do
            routing.output[i] --> portIf[i].fromHigherLayer;
            routing.input[i] <-- portIf[i].toHigherLayer;
            portIf[i].toPort --> outputPorts[i];
            portIf[i].fromPort <-- inputPorts[i];
        endfor;
        routing.output[numOfPorts] --> localUser.input;
        routing.input[numOfPorts] <-- localUser.output;
endmodule
```

Example 2: Chain

For example, one can create a chain of modules like this:

```
module Chain
    parameters: count: const;
    submodules:
        node : Node [count]
            gatesizes:
                in[2], out[2];
            gatesizes if index==0 || index==count-1:
                in[1], out[1];
    connections:
        for i = 0..count-2 do
            node[i].out[i!=0 ? 1 : 0] --> node[i+1].in[0];
            node[i].in[i!=0 ? 1 : 0] <-- node[i+1].out[0];
        endfor;
endmodule
```

Example 3: Binary Tree

One can use conditional connections to build a binary tree. The following NED code loops through all possible node pairs, and creates the connections needed for a binary tree.

```
simple BinaryTreeNode
    gates:
        in: fromupper;
        out: downleft;
        out: downright;
endsimple
```

```
module BinaryTree
    parameters:
        height: const;
    submodules:
        node: BinaryTreeNode [ 2^height-1 ];
    connections nocheck:
        for i = 0..2^height-2, j = 0..2^height-2 do
            node[i].downleft --> node[j].fromupper if j==2*i+1;
            node[i].downright --> node[j].fromupper if j==2*i+2;
        endfor;
endmodule
```

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 `nocheck` modifier. Consequently, it is the simple modules' responsibility not to send on a gate which is not leading anywhere.

An alert reader might notice that there is a better alternative to the above code. Each node except the ones at the lowest level of the tree has to be connected to exactly two nodes, so we can use a single loop to create the connections.

```
module BinaryTree2
    parameters:
        height: const;
    submodules:
        node: BinaryTreeNode [ 2^height-1 ];
    connections nocheck:
        for i=0..2^(height-1)-2 do
            node[i].downleft --> node[2*i+1].fromupper;
            node[i].downright --> node[2*i+2].fromupper;
        endfor;
endmodule
```

Example 4: Random graph

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

```
module RandomGraph
    parameters:
        count: const,
        connectedness; // 0.0<x<1.0
    submodules:
        node: Node [count];
            gatesizes: in[count], out[count];
    connections nocheck:
        for i=0..count-1, j=0..count-1 do
            node[i].out[j] --> node[j].in[i]
                if i!=j && uniform(0,1)<connectedness;
        endfor;
endmodule
```

Note the use of the `nocheck` modifier here too, to turn off error messages given by the network setup code for unconnected gates.

4.9.2 Design patterns for compound modules

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 do
    node[i].out[...] --> node[j].in[...] if condition(i,j);
endfor;
```

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 do
    node[i].out[j] --> node[rightNodeIndex(i,j)].in[j];
endfor;
```

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 do
    node[leftNodeIndex(i)].out[...] --> node[rightNodeIndex(i)].in[...];
endfor;
```

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

The Serial 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 specifying constant submodule/gate vector sizes and explicitly listing all connections, like you would do it in most existing simulators.

4.9.3 Topology templates

Overview

Topology templates are nothing more than compound modules where one or more submodule types are left as parameters (using the `like` phrase of the NED language). You can write such modules which implement mesh, hypercube, butterfly, perfect shuffle or other topologies, and you can use them wherever needed in you simulations. With topology templates, you can reuse *interconnection structure*.

An example: hypercube

The concept is demonstrated on a network with hypercube interconnection. When building an N-dimension hypercube, we can exploit the fact that each node is connected to N others which differ from it only in one bit of the binary representations of the node indices (see Fig. 4.2).

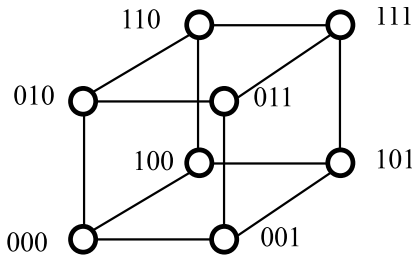

Figure 4.2: Hypercube topology

The hypercube topology template is the following (it can be placed into a separate file, e.g `hypercube.ned`):

```
simple Node
    gates:
        out: out[];
        in: in[];
endsimple

module Hypercube
    parameters:
        dim, nodetype;
    submodules:
        node: nodetype[2^dim] like Node
        gatesizes:
            out[dim], in[dim];
    connections:
        for i=0..2^dim-1, j=0..dim-1 do
            node[i].out[j] --> node[i # 2^j].in[j]; // # is bitwise XOR
```

```
        endfor;
endmodule
```

When you create an actual hypercube, you substitute the name of an existing module type (e.g. `"Hypercube_PE"`) for the nodetype parameter. The module type implements the algorithm the user wants to simulate and it must have the same gates that the Node type has. The topology template code can be used through importing the file:

```
import "hypercube.ned";

simple Hypercube_PE
    gates: out: out[]; in: in[];
endsimple

network hypercube: Hypercube
    parameters:
        dim = 4,
        nodetype = "Hypercube_PE";
endnetwork
```

If you put the nodetype parameter to the ini file, you can use the same simulation model to test e.g. several routing algorithms in a hypercube, each algorithm implemented with a different simple module type – you just have to supply different values to nodetype, such as `"WormholeRoutingNode"`, `"DeflectionRoutingNode"`, etc.

4.10 Large networks

There are situations when using hand-written NED files to describe network topology is inconvenient, for example when the topology information comes from an external source like a network management program.

In such case, you have two possibilities:

1. generating NED files from data files
2. building the network from C++ code

The two solutions have different advantages and disadvantages. The first is more useful in the model development phase, while the second one is better for writing larger scale, more productized simulation programs. In the next sections we examine both methods.

4.10.1 Generating NED files

Text processing programs like `awk` or `perl` are excellent tools to read in textual data files and generate NED files from them. Perl also has extensions to access SQL databases, so it can also be used if the network topology is stored in a database.

The advantage is that the necessary `awk` or `perl` program can be written in a releatively short time, and it is inexpensive to maintain afterwards: if the structure of the data files change, the NED-creating program can be easily modified. The disadvantage is that the resulting NED files are often quite big and the C++ compilation of the `*_n.cc` files may take long.

This method is best suited in the first phase of a simulation project when the topology, the format of the data files, etc. have not yet stabilized.

4.10.2 Building the network from C++ code

Another alternative is to write C++ code which becomes part of the simulation executable. The code would read the topology data from data files or a database, and build the network directly, using dynamic module creation (to be described later, in section 5.10). The code which builds the network would be similar to the `*_n.cc` files output by nedc.

Since writing such code is more complex than letting perl generate NED files, this method is recommended when the simulation program has to be somewhat more productized, for example when OMNeT++ and the simulation model is embedded into a larger program, e.g. a network design tool.

4.11 XML support

Future OMNeT++ versions will contain strong XML support. NED files will have XML representations, and the two forms can be freely converted to each other. XML is much more suited for machine processing, e.g. XSLT can be used to produce NED via XML, extract information from NED files, generating HTML documentation from NED, etc.

The current version (2.3) contains an alpha version of these tools. *nedtool* is going to replace nedc, and it offers much more functionality.

Converting a NED file to XML:

```
nedtool -x wireless.ned
```

It generates `wireless_n.xml`. Several switches control the exact content and details of the resulting XML as well as the amount of checks made on the input.

Converting the XML representation back to NED:

```
nedtool -n wireless.xml
```

The result is `wireless_n.ned`.

Using nedtool as nedc to generate C++ code:

```
nedtool wireless.ned
```

The resulting code is more compact and less redundant than the one created by nedc. As a result, nedtool-created `_n.cc` C++ files compile much faster.

You can generate C++ code from the XML format, too:

```
nedtool wireless.xml
```

The opp_nedtool command uses XSLT to produce HTML documentation from the given NED files, much like Javadoc or Doxygen. [4]

```
opp_neddoc *.ned
```

The output file is called `neddoc.html`. opp_nedtool can be very useful in discovering and understanding the structure of large models like the IP-Suite. On Unix, you'd use it like this:

[4] You're going to need xsltproc (part of libxml/libxslt) installed on your system. Since Gnome and KDE also heavily rely on these components, there's a good chance it is already present on your system.

```
cd IPSuite
opp_neddoc ‘find . -name *.ned‘
```

The HTML output is currently rather plain, but it is going to be improved.

4.12 GNED – Graphical NED Editor

The GNED editor allows you to design compound modules graphically. GNED works with NED files – it doesn't use any nasty internal file format. You can load any of your existing NED files, edit the compound modules in it graphically and then save the file back. The rest of the stuff in the NED file (simple modules, channels, networks etc.) will survive the operation. GNED puts all graphics-related data into display strings.

GNED works by parsing your NED file into an internal data structure, and regenerating the NED text when you save the file. One consequence of this is that indentation will be "canonized" – hopefully you consider this fact as a plus and not as a minus. Comments in the original NED are preserved – the parser associates them with the NED elements they belong to, so comments won't be messed up even if you edit the graphical representation to death by removing/adding submodules, gates, parameters, connections, etc.

GNED is now a fully two-way visual tool. While editing the graphics, you can always switch to NED source view, edit in there and switch back to graphics. Your changes in the NED source will be immediately backparsed to graphics; in fact, the graphics will be totally reconstructed from the NED source and the display strings in it.

GNED is still under development. There are some missing functions and bugs, but overall it should be fairly reliable. See the TODO file in the GNED source directory for problems and missing features.

4.12.1 Comment parsing

It is useful to know how exactly GNED identifies the comments in the NED file. The following (maybe a bit long) NED code should explain it:

```
// ----------------------------------------------------------------
// File: sample.ned
//
// This is a file comment. File comments reach from the top of
// the file till the last blank line above the first code line.
// ----------------------------------------------------------------
//

// The file comment can also contain blank lines, so this is
// still part of the above file comment.
//
// Module1 --
//
// This is a banner comment for the Module1 declaration below.
// Banner comments can be multi-line, but they are not supposed
// to contain blank lines. (Otherwise the lines above the blank
// one will be taken as part of a file comment or trailing comment.)
//
module Module1
```

```
    submodules: // and this is right-comment
        // This is another banner comment, for the submodule
        submod1: Module;
            display: "p=120,108;b=96,72,rect";
            connections:
                out --> submod1.in; // Right-comments can also be
                                    // multi-line.
endmodule

// Finally, this is a trailing comment, belonging to the above
// module. It may contain blank lines. Trailing comments are
// mostly used to put separator lines into the file, like this:
// ----------------------------------------------------------------
// Module2 --
//
// an empty module
//
module Module2
endmodule
```

4.12.2 Keyboard and mouse bindings

In graphics view, there are two editing modes: draw and select/mode. The mouse bindings are the following:

Mouse	Effect
In *draw* mode:	
Drag out a rectangle in empty area:	create new submodule
Drag from one submodule to another:	create new connection
Click in empty area:	switch to select/move mode
In *select*/*move* mode:	
Click submodule/connection:	select it
Ctrl-click submodule/conn.:	add to selection
Click in empty area:	clear selection
Drag a selected object:	move selected objects
Drag submodule or connection:	move it
Drag either end of connection:	move that end
Drag corner of (sub)module:	resize module
Drag starting in empty area:	select enclosed submodules/connections
Del key	delete selected objects
Both editing modes:	
Right-click on module/submodule/connection:	popup menu
Double-click on submodule:	go into submodule
Click name label	edit name
Drag&drop module type from the tree view to the canvas	create a submodule of that type

Chapter 5

Simple Modules

The activities of simple modules are implemented by the user. The algorithms are programmed in C++, using the OMNeT++ class library. The following sections contain a short introduction to discrete event simulation in general, how its concepts are implemented in OMNeT++, and gives an overview and practical advice on how to design and code simple modules.

5.1 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. If you're familiar with DES, you can skip the next few sections.

5.1.1 Discrete Event Simulation

A *Discrete Event System* is a system where state changes (events) happen at discrete points of 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. (Continuous systems are modelled using differential equations and suchlike.)

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 or which refers to how long the simulation program has been running or how much CPU time it has consumed.

5.1.2 The event loop

Discrete event simulations maintain 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 be quite different 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 cancelling timeouts.

Simulation stops when there are no more 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 simulation programmer will typically want to record statistics into output files.

5.1.3 Simple modules in OMNeT++

In OMNeT++, events occur inside simple modules. Simple modules encapsulate C++ code that generate and react to events, in other words, implement 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 stuffed 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 `activity()`
- void `handleMessage(cMessage *msg)`
- 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 `activity()` and `handleMessage()` functions are called during event processing. This means that the user will implement the model's behavior in these functions. `activity()` and `handleMessage()` implement different event processing strategies: for each simple module, the user has to redefine exactly one of these functions. `activity()` is a coroutine-based solution which implements the process interaction approach (coroutines are non-preemptive [cooperative] threads). `handleMessage()` is a method that is called by the simulation kernel when the module receives a message. 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.

All these functions will be discussed later in detail.

5.1.4 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 with the module sending a message to itself.

Simulation time in OMNeT++ is stored in the C++ type `simtime_t`, which is a typedef for `double`.

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

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

Priority is a user-assigned integer attribute of messages.

Storing simulation time in doubles may sometimes cause inconveniences. Due to finite machine precision, two doubles calculated in two different ways do not always compare equal even if they mathematically should be. For example, addition is not an associative operation when it comes to floating point calculations: $(x + y) + z! = x + (y + z)!$ (See [Gol91]). This means that if you want to explicitly rely on the arrival times of two events being the same,

you should take care that they are calculated in exactly the same way. Another possible approach is to avoid equal arrival times, for example by adding/subtracting small values to schedule times to ensure specific execution order (*inorder_epsilon*).

One may suggest introducing a small *simtime_precision* parameter in the simulation kernel that would force t_1 and t_2 to be regarded equal if they are "very close" (if they differ less than *simtime_precision*). However, in addition to the problem determining the correct value for *simtime_precision*, this approach is likely to cause confusion in many cases.

5.1.5 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 it.

5.2 Packet transmission modeling

5.2.1 Delay, bit error rate, data 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 is the probability that a bit is incorrectly transmitted. Thus, the probability that a message of *n* bits length is transferred correctly is:

$P(\textit{sent message received properly}) = (1 - \textit{ber})^n$
where *ber* = bit error rate and *n* = number of bits in message.

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. 5.1).

The above model is not applicable for modeling some protocols like Token Ring and FDDI where the stations repeat the bits of a frame that arrives on the ring immediately, without waiting for the whole frame to arrive; in other words, frames "flow through" the stations,

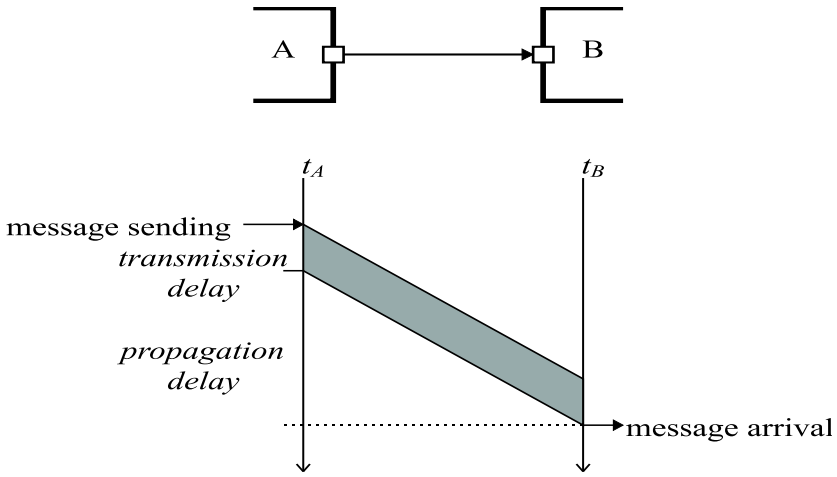

Figure 5.1: Message transmission

being delayed only a few bits. If you want to model such networks, the data rate modeling feature of OMNeT++ cannot be used.

If a message travels along a route, through successive links and compound modules, the model behaves as if each module waited until the last bit of the message arrives and only start its transmission then (Fig. 5.2).

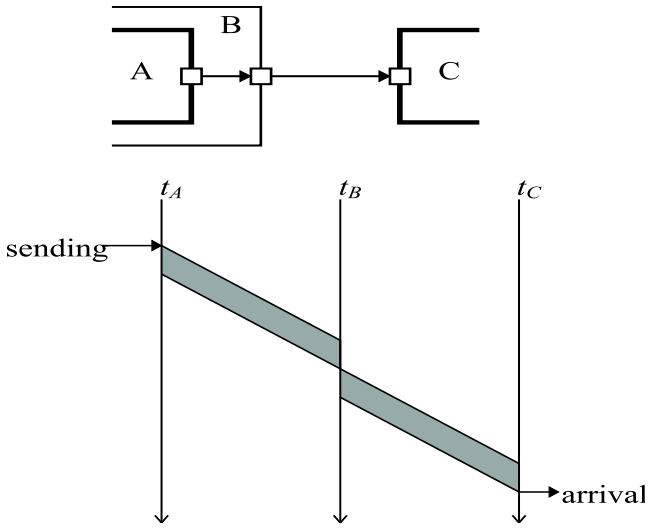

Figure 5.2: Message sending over multiple channels

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

5.2.2 Multiple transmissions on links

If data rate is specified for a connection, a message will have a certain nonzero transmission time, depending on its length. This means that when a message is sent out through an output gate, the message "reserves" the gate for a given period ("it is being transmitted").

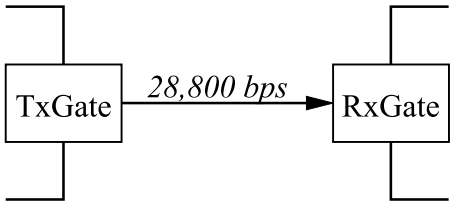

Figure 5.3: Connection with a data rate

While a message is under transmission, other messages have to wait until the transmission finishes. You can still send messages while the gate is busy, but the beginning of the modelled message transmission will be delayed, just like the gate had an internal queue for the messages waiting to be transmitted.

The OMNeT++ class library provides you with functions to check whether a certain output gate is transmitting or to learn when it finishes transmission.

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

Implementation of message sending

Message sending is implemented in the following way: the arrival time and the bit error flag of a message are calculated at once, when the `send()` (or similar) function is invoked. That is, if the message travels through several links until it reaches its destination, it is *not* scheduled individually for each link, but rather, every calculation is done once, within the `send()` call. This implementation was chosen because of its run-time efficiency.

In the actual implementation of queuing the messages at busy gates and modeling the transmission delay, messages do not actually queue up in gates; gates do not have internal queues. Instead, as the time when each gate will finish transmission is known at the time of sending the message, the arrival time of the message can be calculated in advance. Then the message will be stored in the event queue (FES) until the simulation time advances to its arrival time and it is retrieved by its destination module.

Consequence

The implementation has the following consequence. If you change the delay (or the bit error rate, or the data rate) of a link during simulation, the modeling of messages sent "just before" the parameter change will not be accurate. Namely, if link parameters change while a message is "under way" in the model, that message will not be affected by the parameter change, although it should. However, all subsequent messages will be modelled correctly. Similar for data rate: if a data rate changes during the simulation, the change will affect only the messages that are *sent* after the change.

If it is important to model gates and channels with changing properties, you can go two ways:

- write sender module such that they schedule events for when the gate finishes its current transmission and send then;
- alternatively, you can implement channels with simple modules ("active channels").

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 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 will signal the arrival of the packet. In some implementations, also output gates have packet queues where packets wait until the channel becomes free (available for transmission).

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

5.3 Defining simple module types

5.3.1 Overview

The C++ implementation of a simple module consists of:

- declaration of the module class: your class subclassed from `cSimpleModule` (either directly or indirectly)
- a module type registration (`Define_Module()` or `Define_Module_Like()` macro)
- implementation of the module class

For example, the C++ source for a Sliding Window Protocol implementation might look like this:

```
// file: swp.cc
#include <omnetpp.h>

// module class declaration:
class SlidingWindow : public cSimpleModule
{
    Module_Class_Members(SlidingWindow,cSimpleModule,8192)
    virtual void activity();
};

// module type registration:
Define_Module( SlidingWindow );

// implementation of the module class:
void SlidingWindow::activity()
{
  int windowSize = par("windowSize");
...
}
```

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

```
// file: swp.ned
simple SlidingWindow
    parameters:
        windowSize: numeric const;
    gates:
        in: fromNet, fromUser;
        out: toNet, toUser;
endsimple
```

5.3.2 The module declaration

The module declaration

- announces that you're going to use the class as a simple module type
- associates the module class with an interface declared in NED

Forms of module declaration

Module declarations can take two forms:

```
Define_Module(classname);
Define_Module_Like(classname, neddeclname);
```

The first form associates the class (subclassed from `cSimpleModule`) with the NED simple module declaration of the same name. For example, the

```
Define_Module(SlidingWindow);
```

line would ensure that when you create an instance of SlidingWindow in your NED files, the module has the parameters and gates given in the simple SlidingWindow NED declaration, and the implementation will be an instance of the SlidingWindow C++ class.

The second form associates the class with a NED simple module declaration of a different name. You can use this form when you have several modules which share the same interface. This feature will be discussed in detail in the next section.

Header files

Module declarations should not be put into header files, because they are macros expanding to lines for which the compiler generates code.

Compound modules

All module types (including compound modules) need to have module declarations. For all compound modules, the NEDC compiler generates the `Define_Module(..)` lines automatically. However, it is your responsibility to put Define_Module(..) lines into one of the C++ sources for all your simple module types.

Implementation

Unless you are dying to learn about the dirty internals, you may just as well skip this section. But if you're interested, here it is: `Define_Module()` (and also `Define_Module_Like()`) is a macro which expands to a function definition plus the definition of a global object, something like this ugly code (luckily, you won't ever need to be interested in it):

```
static cModule *MyClass__create(const char *name, cModule *parentmod)
    return (cModule *) new MyClass(name, parentmod);

cModuleType MyClass__type("MyClass","MyClass",
 (ModuleCreateFunc)MyClass__create);
```

The `cModuleType` object can act as a factory: it is able to create an instance of the given module type. This, together with the fact that all `cModuleType` objects are available in a single linked list, allows OMNeT++ to instantiate module types given only their class names as strings, without having to include the class declaration into any other C++ source.

The global object also stores the name of the NED interface associated with the module class. The interface description object (another object, generated by nedc) is looked up automatically at network construction time. Whenever a module of the given type is created, it will automatically have the parameters and gates specified in the associated interface description.

5.3.3 Several modules, single NED interface

To support submodule types defined as parameters in NED files (see section 4.5.3), you can reuse an existing NED simple module definition for several simple module types.

Suppose you have three different C++ module classes (`TokenRingMAC`, `EthernetMAC`, `FDDIMAC`) which have identical gates and parameters. Then you can create a single NED declaration, `GenericMAC` for them and write the following module declarations in the C++ code:

```
Define_Module_Like(TokenRingMAC, GenericMAC);
Define_Module_Like(EthernetMAC, GenericMAC);
Define_Module_Like(FDDIMAC, GenericMAC);
```

You won't be able to directly refer to the `TokenRingMAC`, `EthernetMAC`, `FDDIMAC` module types in your NED files, because NED doesn't know about them (their names don't appear in any NED file you could import), but you can use them wherever a submodule type was defined as a parameter to the compound module:

```
module Host
    parameters:
        macType: string;
    submodules:
        mac: macType like GenericMAC;
                // if macType=="EthernetMAC" --> OK!
     ...
endmodule
```

The `macType` parameter should take the value `"TokenRingMAC"`, `"EthernetMAC"` or `"FDDIMAC"`, and a submodule of the appropriate type will be created. The value for the parameter can even be given in the ini file. This gives you a powerful tool to customize simulation models (see also *Topology templates*, Section 4.9.3).

5.3.4 The class declaration

As mentioned before, simple module classes have to be derived from `cSimpleModule` (either directly or indirectly). In addition to overwriting some of the previously mentioned four member functions (`initialize()`, `activity()`, `handleMessage()`,`finish()`), you have to write a constructor and some more functions. Some of this task can be automated, so when writing the C++ class declaration, you have two choices:

1. either use a macro which expands to the "stock" version of the functions
2. or write them yourself.

Using macro to declare the constructor

If you choose the first solution, you use the `Module_Class_Members()` macro:

```
Module_Class_Members(classname, baseclass, stacksize);
```

The first two arguments are obvious (*baseclass* is usually `cSimpleModule`), but *stacksize* needs some explanation. If you use `activity()`, the module code runs as a coroutine, so it will need a separate stack. (This will be discussed in detail later.)

As an example, the class declaration

```
class SlidingWindow : public cSimpleModule
{
    Module_Class_Members(SlidingWindow,cSimpleModule,8192)
    ...
};
```

expands to something like this:

```
class SlidingWindow : public cSimpleModule
{
  public:
    SlidingWindow(const char *name, cModule *parentmodule,
        unsigned stacksize = 8192) :
        cSimpleModule(name, parentmodule, stacksize) {}
    ...
};
```

Expanded form of the constructor

If you have data members in the class that you want to initialize in the constructor, you cannot use the `Module_Class_Members()` macro. Then you have to write the constructor yourself.

The constructor should take the following arguments (which you also have to pass further to the base class):

- `const char *name`, which is the name of the module
- `cModule *parentmodule`, pointer to the parent module

- `unsigned stacksize=`*`stacksize`*, the coroutine stack size

You should not change the number or types of the arguments taken by the constructor, because it will be called by OMNeT++-generated code.

An example:

```
class TokenRingMAC : public cSimpleModule
{
  public:
    cQueue queue; // a data member
    TokenRingMAC(const char *name, cModule *parentmodule,
                 unsigned stacksize = 8192);
    ...
};

TokenRingMAC(const char *name, cModule *parentmodule,
    unsigned stacksize) :
    cSimpleModule(name, parentmodule, stacksize), queue("queue")
{
  // initialize data members
}
```

Stack size decides between activity() and handleMessage()

- if the specified stack size is zero, `handleMessage()` will be used;
- if it is greater than zero, `activity()` will be used.

If you make a mistake (e.g. you forget to set zero stack size for a `handleMessage()` simple module): the default versions of the functions issue error messages telling you what is the problem.

5.3.5 Decomposing activity()/handleMessage()

It is usually a good idea to decompose a `activity()` or `handleMessage()` function when it grows too large. "Too large" is a matter of taste of course, but you should definitely consider splitting up the function if it is more that a few screens (say 50-100 lines) long. This will have a couple of advantages:

- will help future readers of the code understand your program;
- will help *you* understand what it is you're really programming and bring some structure into it;
- will enable you to customize the class by inheriting from it and overwriting member functions

If you have variables which you want to access from all member functions (typically state variables are like that), you'll need to add those variables to the class as data members.

Let's see an example:

```
class TransportProtocol : public cSimpleModule
{
  public:
    Module_Class_Members(TransportProtocol, cSimpleModule, 8192)
    int windowSize;
    int n_s; // N(s)
    int n_r; // N(r)
    cOutVector eedVector;
    cStdDev eedStats;
    //...

    virtual void activity();
    virtual void recalculateTimeout();
    virtual void insertPacketIntoBuffer(cMessage *packet);
    virtual void resendPacket(cMessage *packet);
    //...
};

Define_Module( TransportProtocol );

void TransportProtocol::activity()
{
    windowSize = par("windowSize");
    n_s = n_r = 0;
    eedVector.setName("End-to-End Delay");
    eedStats.setName("eedStats");
    //...
}

//...
```

Note that you may have to use the expanded form of the constructor (instead of `Module_Class_Members()`) to pass arguments to the constructors of member objects like eedVector and eedStats. But most often you don't need to go as far as that; for example, you can set parameters later from `activity()`, as shown in the example above.

5.3.6 Using inheritance

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 AdvancedTransportProtocol : public TransportProtocol
{
  public:
    Module_Class_Members(AdvancedTransportProtocol, TransportProtocol,
                         8192)
    virtual void recalculateTimeout();
};

Define_Module( AdvancedTransportProtocol );
```

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

5.3.7 Global variables

If possible, avoid them. Do not use global variables, including static class members. There are several problems with them.

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.

5.4 Adding functionality to cSimpleModule

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

5.4.1 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

One area where the process-style description is especially convenient is 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 want to handle several connections simultaneously, you may dynamically create as instances of the simple module above as needed. Dynamic module creation will be discussed later.

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 corou-

tine 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 another coroutines. This implies that each coroutine must have an 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. But it is important to understand 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 that's 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

All the simulation programmer needs to care about coroutines is to choose the processor stack size for them. This cannot be automated (Eerrr... at least not without hardware support, some trick with virtual memory handling).

16 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-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 it 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()`.

However, you need `finish()` if you want to write statistics at the end of the simulation. And because `finish()` cannot access the local variables of `activity()`, you have to put the variables and objects that contain 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 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
}
```

Advantages and drawbacks of `activity()` vs `handleMessage()`

Advantages:

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

Drawbacks:

- memory overhead: stack allocation may 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

Other simulators

Coroutines are used by a number of other simulation packages:

- All simulation software which inherit from SIMULA (e.g. C++SIM) are 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 on with "normal" preemptive threads). The philosophy is quite similar to OMNeT++. PARSEC, being "just" a programming language, has a more elegant syntax but much less features than OMNeT++.
- Many Java-based simulation libraries are based on Java threads.

5.4.2 handleMessage()

Function called for each event

The idea is that at each event we simply call a user-defined function instead of switching to a coroutine that has `activity()` running in it. The "user-defined function" is the `handleMessage(cMessage *msg)` virtual member function of `cSimpleModule`; the user has to redefine the function to make it do useful work. Calls to `handleMessage()` occur in the main stack of the program – no coroutine stack is needed and no context switch is done.

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 pseudocode of the event loop which is able to handle both `activity()` and `handleMessage()` simple modules:

```
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 then.

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 used in the normal way: to record statistics information accumulated in data members of the class at the end of the simulation.

Application area

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

1. When you expect the module to be used in large simulations, involving several thousand modules. In such cases, the module stacks required by `activity()` would simply consume too much memory.
2. For modules which maintain little or no state information, such as packet sinks, `handleMessage()` is more convenient to program.
3. Other good candidates are modules with a large state space and many arbitrary state transition possibilities (i.e. where there are many possible subsequent states for any state). Such algorithms are difficult to program with `activity()`, or the result is code which is better suited for `handleMessage()` (see rule of thumb below). Most communication protocols are like this.

In general, if your `activity()` function contains no `wait()` and it has only one `receive()` call at the top of an infinite loop (`while(true)` or `for(;;)`), you can trivially convert it to `handleMessage()`. The body of the infinite loop becomes the body to `handleMessage()`, state variables inside `activity()` become data members in the module class, and you initialize them in `initialize()`.

That is, the following code:

```
activity()
{
    initialization code
    while(true)
    {
        msg = receive();
        // code which doesn't contain
        // receive() or wait() calls
    }
}
```

becomes like this:

```
initialize()
{
    initialization code
}

handleMessage( msg )
{
    // code which doesn't contain
    // receive() or wait() calls
}
```

Example 1: Simple traffic generators and sinks

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

```
PacketGenerator::handleMessage(m)
{
    create and send out packet
    schedule m again to trigger next call to handleMessage
      // (self-message)
}
PacketSink::handleMessage(m)
{
    delete m
}
```

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
{
    Module_Class_Members(Generator,cSimpleModule,0)
    // note zero stack size!
    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 2: 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
{
    Module_Class_Members(Generator,cSimpleModule,0)
    // note the zero stack size!
    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);
    }
}
```

Advantages and drawbacks of `handleMessage()` vs `activity()`

Advantages:

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

Drawbacks:

- local variables cannot be used to store state information
- need to redefine `initialize()`
- programming model is inconvenient in some cases

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$^{(TM)}$ (MIL3, Inc.) 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.

5.4.3 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 has its `initialize()` function called *before* all its submodules have.

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 as with `initialize()`: first submodules, then the containing compound module. (The bottom line is that in the moment there's no "official" possibility to redefine `initialize()` and `finish()` for compound modules; the unofficial way is to write into the nedc-generated C++ code. Future versions of OMNeT++ will support adding these functions to compound modules.)

This is summarized in the following pseudocode (although you won't find this code "as is" in the simulation kernel sources):

```
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. For example, 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 stuff which are to run only on successful completion.

Cleanup code should go into the destructor. But in fact, you almost never need to write a destructor because OMNeT++ keeps track of objects you create and disposes of them automatically (sort of automatic garbage collection). However it cannot track objects not derived from `cObject`, so they may need to be deleted manually from the destructor. Garbage collection is discussed in more detail in section 7.13.5.

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. [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 many simulators (e.g. OPNET) 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 fact is also evidenced 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).

5.5 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 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.

[1] 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 if effect and return 1.

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 tell 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 parens 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,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).name()
      << ((exiting) ? ": leaving state " : ": entering state ")
      << (fsm).stateName() << 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 ugly, but if you're dying to see it, it's 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 flavour of a bursty generator. It has 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
{
 public:
  Module_Class_Members(BurstyGenerator,cSimpleModule,0);

  // 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("sleep_time_mean");
  burstTimeMean = par("burst_time_mean");
  sendIATime = par("send_ia_time");
  msgLength = &par("msg_length");
  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->setLength( (long) *msgLength );
      job->setTimestamp();
      send( job, "out" );
      // return to ACTIVE
      FSM_Goto(fsm,ACTIVE);
      break;
    }
  }
}
```

5.6 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 6. 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.

5.6.1 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);
```

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 function uses the gate Id, and because it does not have to search through the gate array, it is faster than the first one.

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("packet"), outGateId);
  wait(5);
}
```

Modeling packet transmissions

If you're sending messages over a link that has (nonzero) data rate, it is modeled in the way that has been described earlier in this manual, in section 5.2.

If you want to have full control over the transmission process, you'll probably need the `isBusy()` and `transmissionFinishes()` member functions of `cGate`. They are described in section 5.8.3.

5.6.2 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 = (cMessage *) msg->dup();
    send(copy, "out", i);
}
delete msg;
```

You might have noticed that copying the message for the last gate is redundant (we could send out the original message), so it can be optimized out like this:

```
for (int i=0; i<n-1; i++)   // note n-1 instead of n
{
    cMessage *copy = (cMessage *) msg->dup();
    send(copy, "out", i);
}
send(msg, "out", n-1);  // send original on last gate
```

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 message, and retain the original. When you're sure there won't be any more retransmission, you can delete the original message.

Creating and sending a copy:

```
// (re)transmit packet:
cMessage *copy = (cMessage *) 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 has full authority over it – it can send it further, 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*. [2]

[2]The feature does not increase runtime overhead significantly, because it uses the object ownership management (described in Section 7.13); it merely checks that the owner of the message is the module that wants to send it.

5.6.3 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( some_delay );
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 5.6.7, “Self-messages”, would need to be employed.

However, there is a more straightforward method than the above two, and this is delayed sending. Delayed sending can be done with one of these functions:

```
sendDelayed(cMessage *msg, double delay, const char *gate_name, int in-
dex);
sendDelayed(cMessage *msg, double delay, int gate_id);
```

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 nonnegative.

Example:

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

5.6.4 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, and it takes the pointer of the remote module (`cModule *`). You can also specify a delay and an input gate of the destination module.

```
cModule *destinationmodule =...;
double delay = truncnormal(0.005, 0.0001);
sendDirect( new cMessage, delay, destinationmodule, "in" );
```

The destination module receives the message as if it was sent “normally”.

5.6.5 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.

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

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

Deprecated functionality: putaside-queue

Earlier versions of OMNeT++ supported something called *putaside-queue* which stored messages that arrived while the module was waiting for something else. For example, a function called `receiveOn()` waited for a message to arrive on a specific gate, and messages that actually arrived on another gate were inserted in the putaside-queue. `receive()` returned messages from the putaside-queue first, and it waited for new messages to arrive only if putaside-queue was empty.

Practice has shown that putaside-queue and associated functionality bore little usefulness, and at the same time it was very confusing to people. Therefore putaside-queue, `receiveOn()`, `receiveNew()`, and `receiveNewOn()` have been deprecated in OMNeT++ 2.3, and will be entirely removed from further releases.

5.6.6 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 module parameter
  wait( par("interArrivalTime") );

  // generate and send message
  ...
}
```

If you expect messages to arrive during the wait period, you can use the `waitAndEnqueue()` function. It takes a pointer to a queue object (of class `cQueue`, described in chapter 7) in addition to the wait interval. Messages that arrive during the wait interval will be accumulated in the queue, so you can process them after the `wait()` call has returned.

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

Change in wait() semantics

The semantics of `wait()` has slightly changed after OMNeT++ 2.3, due to the deprecation of the putaside-queue (see section 5.6.5). Up to release 2.3, messages that arrived during the wait interval were accumulated in the putaside-queue. In later releases, it will be a runtime error if a message arrives during the wait interval – if you want to allow such messages, you should use `waitAndEnqueue()`.

5.6.7 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 sending 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);
}
```

5.6.8 Stopping the simulation

Normal termination

You can finish the simulation with the `endSimulation()` function:

```
endSimulation();
```

However, typically you don't need `endSimulation()` because you can specify simulation time and CPU time limits in the ini file (see later).

Stopping on errors

If your simulation detects an error condition and wants to stop the simulation, you can do it with the `error()` member function of `cModule`. It is used like `printf()`:

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

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

5.7 Accessing module parameters

Module parameters can be accessed with the `par()` member function of `cModule`:

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

The `cPar` class is a general value-storing object. It supports type casts to numeric types, so parameter values can be read like this:

```
int num_tasks = par("num_tasks");
double proc_delay = par("proc_delay");
```

If the parameter is a random variable or its value can change during execution, it is best to store a reference to it and re-read the value each time it is needed:

```
cPar& wait_time = par("wait_time");
for(;;)
{
  //...
  wait( (simtime_t)wait_time );
}
```

If the wait_time parameter was given a random value (e.g. `exponential(1.0)`) in the NED source or the ini file, the above code results in a different delay each time.

Parameter values can also be changed from the program, during execution. If the parameter was taken by reference (with a `ref` modifier in the NED file), other modules will also see the change. Thus, parameters taken by reference can be used as a means of module communication.

An example:

```
par("wait_time") = 0.12;
```

Or:

```
cPar& wait_time = par("wait_time");
wait_time = 0.12;
```

See `cPar` explanation later in this manual for further information on how to change a `cPar`'s value.

5.8 Accessing gates and connections

5.8.1 Gate objects

Module gates are `cGate` objects. Gate objects know whether and to which gate they are connected, and they can be asked about the parameters of the link (delay, data rate, etc.)

The `gate()` member function of `cModule` returns a pointer to a `cGate` object, and an overloaded form of the function lets you to access elements of a vector gate:

```
cGate *outgate = gate("out");
cGate *outvec5gate = gate("outvec",5);
```

For gate vectors, the first form returns the first gate in the vector (at index 0).

The `isVector()` member function can be used to determine if a gate belongs to a gate vector or not. But this is almost insignificant, because non-vector gates are treated as vectors with size 1.

Given a gate pointer, you can use the `size()` and `index()` member functions of `cGate` to determine the size of the gate vector and the index of the gate within the vector:

```
int size2 = outvec5gate->size(); // --> size of outvec[]
int index = outvec5gate->index(); // --> 5 (it is gate 5 in the vector)
```

For non-vector gates, `size()` returns 1 and `index()` returns 0.

The `type()` member function returns a character, 'I' for input gates and 'O' for output gates:

```
char type = outgate->type() // --> 'O'
```

Gate IDs

Module gates (input and output, single and vector) are stored in an array within their modules. The gate's position in the array is called the *gate ID*. The gate ID is returned by the `id()` member function:

```
int id = outgate->id();
```

For a module with input gates from_app and in[3] and output gates of to_app and status, the array may look like this:

ID	dir	name[index]
0	*input*	from_app
1	*output*	to_app
2	*empty*	
3	*input*	in[0]
4	*input*	in[1]
5	*input*	in[2]
6	*output*	status

The array may have empty slots. Gate vectors are guaranteed to occupy contiguous IDs, thus it is legal to calculate the ID of *gate[k]* as `gate("gate",0).id()+k`.

Message sending and receiving functions accept both gate names and gate IDs; the functions using gate IDs are a bit faster. Gate IDs do not change during execution, so it is often worth retrieving them in advance and using them instead of gate names.

You can also obtain gate IDs with the `findGate()` member of `cModule`:

```
int id1 = findGate("out");
int id2 = findGate("outvect",5);
```

5.8.2 Link parameters

The following member functions return the link attributes:

```
cLinkType *link = outgate->link();
cPar *d = outgate->delay();
cPar *e = outgate->error();
cPar *r = outgate->datarate();
```

5.8.3 Transmission state

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

The semantics has been described in section 5.2.

An example:

```
cMessage *packet = new cMessage("DATA");
packet->setLength( 1000 );

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

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

```
if (gate("mygate")->toGate()->isBusy())
  //...
```

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

5.8.4 Connectivity

The `isConnected()` member function returns whether the gate is connected. If the gate is an output gate, the gate to which it is connected is obtained by the `toGate()` member function. For input gates, the function is `fromGate()`.

```
cGate *gate = gate("somegate");
if (gate->isConnected())
{
  cGate *othergate = (gate->type()=='O') ?
                     gate->toGate() : gate->fromGate();

  ev << "gate is connected to: " << othergate->fullPath() << endl;
}
else
{
  ev << "gate not connected" << endl;
}
```

An alternative to `isConnected()` is to check the return value of `toGate()` or `fromGate()`. The following code is fully equivalent to the one above:

```
cGate *gate = gate("somegate");
cGate *othergate = (gate->type()=='O') ?
                   gate->toGate() : gate->fromGate();
if (othergate)
  ev << "gate is connected to: " << othergate->fullPath() << endl;
else
  ev << "gate not connected" << endl;
```

To find out to which simple module a given output gate leads finally, you would have to walk along the path like this (the `ownerModule()` member function returns the module to which the gate belongs):

```
cGate *gate = gate("out");
while (gate->toGate()!=NULL)
{
  gate = gate->toGate();
}

cModule *destmod = gate->ownerModule();
```

but luckily, there are two convenience functions which do that: `sourceGate()` and `destinationGate()`.

5.9 Walking the module hierarchy

Module vectors

If a module is part of a module vector, the `index()` and `size()` member functions can be used to query its index and the vector size:

```
ev << "This is module [" << module->index() <<
      "] in a vector of size [" << module->size() << "].\n";
```

Module IDs

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

```
int myModuleId = id();
```

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.module( 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 `parentModule()` member function:

```
cModule *parent = parentModule();
```

For example, the parameters of the parent module are accessed like this:

```
double timeout = parentModule()->par( "timeout" );
```

`cModule`'s `findSubmodule()` and `submodule()` 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->submodule("child",5);
```

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

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

would give the same results as

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

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

The `cSimulation::moduleByPath()` 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(*parentModule()); !iter.end(); iter++)
{
  ev << iter()->fullName();
}
```

(`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 `name()` function returns the same for all modules:

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

Walking along links

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

```
cModule *neighbour = gate( "outputgate" )->toGate()->ownerModule();
```

For input gates, you would use `fromGate()` instead of `toGate()`.

5.10 Dynamic module creation

5.10.1 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 dymically 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.

5.10.2 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 `De-`

`fine_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's 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.

5.10.3 Creating modules

The first step, finding the factory object:

```
cModuleType *moduleType = findModuleType("TCPConnectionHandler");
```

Simplified form

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

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

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

This method can be used for both simple and compound modules. However, its applicability is somewhat limited: 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:

1. find factory object
2. create module
3. set up parameters and gate sizes (if needed)

4. call function that builds out submodules and finalizes the module
5. call function that creates activation message(s) for the new simple module(s)

Each step (except for Step 3.) can be done with one line of code.

See the following example, where Step 3 is omitted:

```
// find factory object
cModuleType *moduleType = findModuleType("TCPConnectionHandler");

// create (possibly compound) module and build its submodules (if any)
cModule *module = moduleType->create( "TCPconn", 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:

```
cModuleType *moduleType = findModuleType("TCP-conn-handler");
cModule *module = moduleType->create( "TCPconn", this );
// set up parameters and gate sizes before we set up its submodules
module->par("window-size") = 4096;
module->setGateSize("to-apps", 3);
module->buildInside();
module->scheduleStart( simTime() );
```

5.10.4 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, if the module was implemented using `activity()`, the `deleteModule()` call does not return to the caller (the reason is that deleting the module also deletes the CPU stack of the coroutine).

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.

5.10.5 Creating connections

Connections can be created using `cGate`'s `connectTo()` method. [3] `connectTo()` should be invoked on the source gate of the connection, and expects the destination gate pointer as an argument:

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

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

The *source* and *destination* words correspond to the direction of the arrow in NED files and which is the same, the direction of messages. That is, you connect an output gate *to* another submodule's input gate; you connect a submodule's output gate *to* the output gate of its parent module; and an input gate of a compound module *to* the input gate of its submodule.

As an example, we create two modules and connect them in both directions:

```
cModuleType *moduleType = findModuleType("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 `cSimpleChannel` 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.

`cSimpleChannel` has `setDelay()`, `setError()` and `setDatarate()` methods to set up the channel attributes. These functions accept pointer to dynamically allocated `cPar` objects. `cPar` will be covered in detail in chapter 7.

An example that sets up a channel with a delay:

```
cSimpleChannel *channel = new cSimpleChannel("channel");

cPar *d = new cPar("delay");
d->setDoubleValue(0.001);
channel->setDelay(d);

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

Chapter 6

Messages

6.1 Messages and packets

6.1.1 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 `cObject` (see section 7.1).
- 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 like remembering 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*, *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->setLength( length );
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.

The values can be obtained by the following functions:

```
int k       = msg->kind();
int p       = msg->priority();
int l       = msg->length();
bool b      = msg->hasBitError();
simtime_t t = msg->timestamp();
```

Duplicating messages

It is often needed to duplicate a message (for example, send one and keep a copy). This can be done in the standard ways as for any other OMNeT++ object:

```
cMessage *copy1 = (cMessage *) msg->dup();
cMessage *copy2 = new cMessage( *msg );
```

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

6.1.2 Message encapsulation

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->setLength(8*2000);
cMessage *tcpseg = new cMessage("tcp");
tcpseg->setLength(8*24);
tcpseg->encapsulate(userdata);
ev << tcpseg->length() << endl; // --> 8*2024 = 16192
```

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

6.1.3 Information about the last sending

There are several variables in `cMessage` that store information about the last time the message was sent or scheduled. These variables can only be read.

`isSelfMessage()` returns true if the message has been scheduled (`scheduleAt()`) as opposed to being sent with one of the `send...()` methods.

```
bool isSelfMessage()
```

The following methods can tell where the message came from and where it arrived (will arrive).

```
int senderModuleId();
int senderGateId();
int arrivalModuleId();
int arrivalGateId();
```

The following two methods are just convenience functions that combine module id and gate id into a gate object pointer.

```
cGate *senderGate();
cGate *arrivalGate();
```

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 creationTime()
simtime_t sendingTime();
simtime_t arrivalTime();
```

6.1.4 Context pointer

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

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

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.

6.1.5 Modeling packets and frames

The `cPacket` class is now deprecated. Please do not use it in new models. [1]

In future OMNeT++ models, the protocol should be represented in the message subclass (i.e. instances of class `IPv6Packet` represent IPv6 packets, and `EthernetFrame` represents Ethernet frames) and/or in the message kind value. PDU is usually represented as a field inside the message class (a protocol header field).

If the former case, the C++ `dynamic_cast` operator can be used to determine if a message object is of a specific protocol:

[1] `cPacket` was a subclass of `cMessage`, and it was originally intended as a base class for packets or frames in a telecommunications network. `cPacket` added two data members: *protocol* and *PDU* (*Protocol Data Unit*, an OSI term). It was envisioned that *procotol* and *PDU* would take their values from globally maintained enums. As it turned out, this was utopistic. In practice, usage of *protocol* was inconsistent (*protocol* and *message kind* played similar roles, creating confusion), and models did not use *PDU* at all. Hence the deprecation of `cPacket`.

```
cMessage *msg = receive();
if (dynamic_cast<IPv6Packet *>(msg) != NULL)
{
    IPv6Packet *ipv6packet = (IPv6Packet *)msg;
    ...
}
```

6.1.6 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 6.2.

Attaching objects

The `cMessage` class has an internal `cArray` object which can carry objects. Only objects that are derived from `cObject` (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 *pklen_distr = new cLongHistogram("pklen_distr");
msg->addObject( pklen_distr );
...
if (msg->hasObject("pklen_distr"))
{
   cLongHistogram *pklen_distr =
       (cLongHistogram *) msg->getObject("pklen_distr");
   ...
}
```

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

You can attach non-object types (or non-`cObject` objects) to the message by using `cPar`'s void* pointer 'P') type (see later in the description of `cPar`). 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 6.2).

The old, deprecated way of adding new fields to messages is via attaching `cPar` objects. There are several downsides of this approach, the worst being being large memory and execution time overhead. `cPar`'s are heavy-weight and fairly complex objects themselves. It has been reported that using `cPar` message parameters might account for a large part of execution time, sometimes as much as 80%. Using `cPar`s is also error-prone because `cPar` 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.

However, if you still need to use cPars, here's a short summary how you can do it. You add a new parameter to the message with the `addPar()` member function, and get back a reference to the parameter object with the `par()` member function. `hasPar()` tells you if the message has a given parameter or not. Message parameters can be accessed also by index in the parameter array. The `findPar()` function returns the index of a parameter or -1 if the parameter cannot be found. The parameter can then be accessed using an overloaded `par()` function.

Example:

```
msg->addPar("dest_addr");
msg->par("dest_addr") = 168;
...
long dest_addr = msg->par("dest_addr");
```

6.2 Message definitions

6.2.1 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 message subclassing feature in OMNeT++ is still somewhat experimental, meaning that:

- The message description syntax and features may slightly change in the future, based on feedback from the community;
- The compiler that translates message descriptions into C++ is a perl script `opp_msgc`. This is a temporary solution until the C++-based `nedtool` is finished.

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
{
    fields:
       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 stuctures.

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 stuctures 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's might be some misunderstanding 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.

6.2.2 Declaring enums

An `enum {..}` 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.

6.2.3 Message declarations

Basic use

You can describe messages with the following syntax:

```
message FooPacket
{
    fields:
        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
{
   fields:
        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
{
  fields:
      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
{
    fields:
        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
{
   fields:
       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
{
   fields:
       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: a string member is different from a character array, which is treated as an array of any other type. For example,

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

will generate the following methods:

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

6.2.4 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 these tasks.

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
{
    fields:
        ...
};
```

For the above example, 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 `cObject`) 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 `cObject` than for those which are not. If there is no `extends`, the generated class will not be derived from `cObject`, thus it will not have `name()`, `className()`, etc. methods. To create a class with those methods, you have to explicitly write `extends cObject`.

```
class MyClass extends cObject
{
    fields:
        ...
};
```

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++ language 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
{
    fields:
        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 above definition, the following code is generated:

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

A struct can have primitive types or other structs are 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:

- field initialization is not supported (it would need constructor)
- struct fields are not initialized to zero (it would need constructor)
- 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
{
   fields:
       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.

6.2.5 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 `cObject` 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 `cObject` 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 so you mislead the message compiler into thinking that your class is rooted in `cObject` when in fact it is not), you'll get a C++ compiler error in the generated header file.

6.2.6 Customizing the generated class

The Generation Gap pattern

Sometimes you need the generated code to 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;
    setLength( 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 following syntax:

```
message FooPacket
{
   properties:
       customize = true;
   fields:
       int payloadLength;
};
```

The `properties` section within the message declaration contains meta-info that affects how generated code will look like. The customize property enables the use of the Generation Gap pattern.

If you process the above code with the message compiler, the generated code will contain a `FooPacket_Base` class instead of `FooPacket`. The idea is that you have to subclass from `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 cObject *dup() {return new FooPacket(*this);}
};

Register_Class(FooPacket);
```

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 contructor, operator=(), dup()
    // ...

    virtual void setPayloadLength(int newlength);
}

void FooPacket::setPayloadLength(int newlength)
{
    // adjust message length
    setLength(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
{
   properties:
       customize = true;
   fields:
       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.

6.2.7 Summary

This section attempts to summarize the possibilities.

You can generate:

- classes rooted in `cObject`
- messages (default base class is `cMessage`)
- classes not rooted in `cObject`
- 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 `cObject`), 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 `enum`s 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 {`
`  properties:`
`  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 (Token Ring, Dyna2, Hypercube) use message definitions. For example, in Dyna2 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

6.2.8 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 the why this is necessary, suppose you manually subclass `cMessage` to get a new message class. You could write the following: [2]

```
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'd notice one drawback of this solution when you try to use Tkenv for debugging. While `cPar`-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 the C++ language 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) {
```

[2] 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.

```
        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 `className()` 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.

Chapter 7

The Simulation Library

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

- events, messages, network packets: the `cMessage` and `cPacket` classes (chapter 6)
- sending and receiving messages, scheduling and cancelling events, terminating the module or the simulation (section 5.6)
- access to module gates and parameters via `cModule` member functions (sections 5.7 and 5.8)
- accessing other modules in the network (section 5.9)
- dynamic module creation (section 5.10)

This chapter discusses the rest of the simulation library:

- random number generation: `normal()`, `exponential()`, etc.
- module parameters: `cPar` class
- storing data in containers: `cArray`, `cQueue`, `cBag` and `cLinkedList` classes
- routing support and discovery of network topology: `cTopology` class
- recording statistics into file: `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()` macro (`cWatch` class)
- sending debug output to and prompting for user input in the graphical user interface (Tkenv): the `ev` object (`cEnvir` class)

7.1 Class library conventions

Base class

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

- name attribute
- `className()` member and other member functions giving textual information about the object
- conventions for assignment, copying, duplicating the object
- ownership control for containers derived from `cObject`
- support for traversing the object tree
- support for inspecting the object in graphical user interfaces (Tkenv)
- support for automatic cleanup (garbage collection) at the end of the simulation

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

Setting and getting attributes

Member functions that set and query object attributes follow consistent naming. The setter member function has the form setSomething(...) and its getter counterpart is named something(), i.e. the "get" verb found in Java and some other libraries is omitted for brevity. For example, the *length* attribute of the `cMessage` class can be set and read like this:

```
msg->setLength( 1024 );
length = msg->length();
```

className()

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

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

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). If you supply a name string, the object will make its own copy (`strdup()`). As an example, you can create a message object like this:

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

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

```
mymsg->setName("mymsg");
```

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

```
const char *name = mymsg->name(); // --> returns ptr to internal copy
                                  // of "mymsg"
```

For convenience and efficiency reasons, the empty string "" and NULL are treated as equivalent by library objects: "" is stored as NULL (so that it does not consume heap), but it is returned as "" (so that it is easier to print out etc). Thus, if you create a message object with either NULL or "" as name, it will be stored as NULL and `name()` will return a pointer to "", a static string:

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

fullName() and fullPath()

Objects have two more member functions which return other sort of names based on the name attribute: `fullName()` and `fullPath()`.

Suppose we have a module in the network university_lan, compound module fddi_ring, simple module station[10]. If you call the functions on the simple module object (`cSimpleModule` inherits from `cObject`, too), the functions will return these values:

```
ev << module->name(); // --> "station"
ev << module->fullName(); // --> "station[10]"
ev << module->fullPath(); // --> "university_lan.fddi_ring.station[10]"
```

These functions work for any object. For example, a local object inside the module would produce results like this:

```
void FDDIStation::activity()
{
    cQueue buffer("buffer");
    ev << buffer->fullPath(); // --> "university_lan.fddi_ring.
                              // station[10].buffer"
}
```

`fullName()` and `fullPath()`, together with `className()` can be used for example to generate informative error messages.

Be aware that `fullName()` and `fullPath()` return pointers to static buffers. Each call will overwrite the previous content of the buffer, so for example you shouldn't put two calls in a single `printf()` statement:

```
ev.printf("object1 is '%s', object2 is '%s'\n",
          object1->fullPath(),
          object2->fullPath()
         ); // WRONG! Same string is printed twice!!!
```

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. `dup()` returns a pointer of type `cObject*`, so it needs to be cast to the proper type:

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

`dup()` works through calling 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. The copying is done properly; object contained in the object will also be duplicated if necessary. For various reasons, `operator=()` does not copy the name string; the copy constructor does it.

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++)
{
    cObject *containedObject = queueIter();
}
```

Ownership control

By default, if a container object is destroyed, it destroys the contained objects too. If you call `dup()`, the contained objects are duplicated too for the new container. This is done so because contained objects are owned by the container; ownership is defined as the right/duty of deallocation. However, there is a fine-grain ownership control mechanism built in which allows you to specify on per-object basis whether you want objects to be owned by the container or not; by calling the `takeOwnership()` member function with false, you tell the container that you don't want it to become the owner of objects that will be inserted in the future.

The ownership mechanism is discussed in detail in section 7.13

7.2 Utilities

Tracing

The tracing feature will be used extensively in the code examples, so it is shortly introduced here. It will be covered in detail in a later section.

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 "
   << seq_num << endl;
```

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

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

Simulation time conversion

There are utility functions which convert simulation time (`simtime_t`) to a printable string (like `"3s 130ms 230us"`) and vica versa.

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

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

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 some rubbish";

simtime_t t = strToSimtime0(s); // now s points to "and some rubbish"
```

Utility <string.h> functions

The `opp_strdup()`, `opp_strcpy()`, `opp_strcmp()` functions are the same as their `<string.h>` equivalents, except that they treat NULL and the empty string (`""`) as identical, and `opp_strdup()` uses operator new instead of `malloc()`.

The `opp_concat()` function might also be useful, for example in constructing object names. It takes up to four `const char *` pointers, concatenates them in a static buffer and returns a pointer to the result. The result's length shouldn't exceed 255 characters.

7.3 Generating random numbers

Random numbers in simulation are never random. Rather, they are produced using deteministic 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. [1]

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. Matematical transformations are used to produce random variates from them that correspond to specific distributions.

[1] There exist real random numbers too, see e.g. http://www.random.org/, http://www.comscire.com, or the Linux /dev/random device.

7.3.1 Random number generators

The current RNG

The currently used random number generator in OMNeT++ is a linear congruential generator (LCG) with a cycle length of $2^{31} - 2$. The startup code of OMNeT++ contains code that checks if the random number generator works OK, so you do not have to worry about this if you port the simulator to a new architecture or use a different compiler.

The random number generator was taken from [Jai91], pp. 441-444,455. It has the following properties:

- Range: $1...2^{31} - 2$
- Period length: $2^{31} - 2$
- Method: $x := (x * 7^5) mod (2^{31} - 1)$
- Verification: if $x[0] = 1$ then $x[10000] = 1,043,618,065$
- Required hardware: exactly 32-bit integer arithmetics

The concrete implementation:

```
long intrand()
{
  const long int a=16807, q=127773, r=2836;
  seed=a*(seed%q) - r*(seed/q);
  if (seed<=0) seed+=INTRAND_MAX;
  return seed;
}
```

Caution!

The above "minimal standard" RNG is only suitable for small-scale simulation studies. As shown by Karl Entacher et al. in [EHW02], the cycle length of about 2^{31} is too small (on todays fast computers it is easy to exhaust all random numbers), and the structure of the generated "random" points is too regular. The [Hel98] paper gives you a broader overview of issues associated with RNGs used for simulation, and it's well worth reading. It also gives you useful links and references to further reading on the topic.

Work is underway to create a flexible and extensible random number architecture in future versions of OMNeT++, and to integrate modern RNGs such as L'Ecuyer's CMRG [LSCK02] with a period of about 2^{191} and/or Mersenne Twister [MN98].

Multiple RNGs

If a simulation program uses random numbers for more than one purpose, the numbers should come from different random number generators. OMNeT++ provides several independent random number generators (by default 32; this number is #defined as `NUM_RANDOM_GENERATORS` in `utils.h`).

To avoid unwanted correlation, it is also important that different simulation runs and different random number sources within one simulation run use non-overlapping series of random numbers, so the generators should be started with seeds well apart. For selecting good seeds, the seedtool program can be used (it is documented later).

Accessing RNGs

Integers can be generated via the `intrand()` function:

```
long rnd = intrand();   // in the range 1..INTRAND_MAX-1
```

The random number seed can be specified in the ini file (`random-seed=`) or set directly from within simple modules with the `randseed()` function:

```
randseed( 10 );         // set seed to 10
long seed = randseed(); // current seed value
```

Zero is not allowed as a seed.

The `intrand()` and `randseed()` functions use generator 0. They have another variant which uses a specified generator:

```
long rnd = genk_intrand(6); // like intrand(), using generator 6
genk_randseed( k, 167 );    // set seed of generator k to 167
```

The `intrand(n)` and `dblrand()` functions are based on `intrand()`:

```
int dice = 1 + intrand(6); // result of intrand(6) is in the range 0..5
                           // (it is calculated as intrand()%6)

double prob = dblrand();   // dblrand() produces numbers in [0,1)
                           // calculated as
                           // intrand()/(double)INTRAND_MAX
```

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

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

7.3.2 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.

7.3.3 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.

7.4 Container classes

7.4.1 Queue class: cQueue

Basic usage

`cQueue` is a container class that acts as a queue. `cQueue` can hold objects of type derived from `cObject` (almost all classes from the 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.

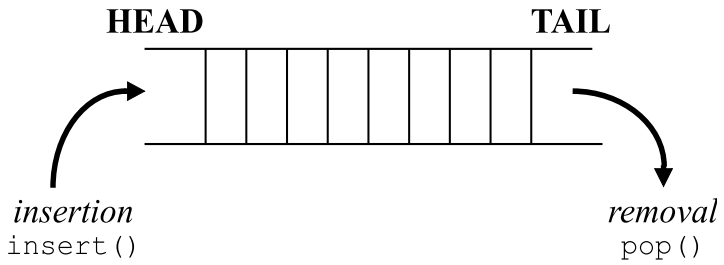

Figure 7.1: What is what with cQueue

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 `tail()` and `head()` functions return pointers to the objects at the tail and head 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 `cObject` 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", cObject::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();
  //...
}
```

7.4.2 Expandable array: cArray

Basic usage

`cArray` is a container class that holds objects derived from `cObject`. `cArray` stores the pointers of the objects inserted instead of making copies. `cArray` works as an array, but if it gets full, it grows automatically. Internally, `cArray` is implemented with an array of pointers; if the array gets full, 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 cPar("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 cPar("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's easy to loop through all the indices with an integer variable. The `items()` member function returns the largest index plus one.

```
for (int i=0; i<array.items(); i++)
{
  if (array[i]) // is this position used?
  {
    cObject *obj = array[i];
    ev << obj->name() << endl;
  }
}
```

7.5 Non-object container classes

There are two container classes to store non-object items: `cLinkedList` and `cBag`. The first one parallels with `cQueue`, the second one with `cArray`. They can be useful if you have to deal with C structs or objects that are not derived from `cObject`.

See the class library reference for more info about them.

7.6 The parameter class: cPar

7.6.1 Basic usage

cPar is a class that is designed to hold a value. The value is numeric (long or double) in the first place, but string, pointer and other types are also supported.

cPar is used in OMNeT++ in the following places:

- as module parameters
- as message parameters

There are many ways to set a cPar's value. One is the set...Value() member functions:

```
cPar pp("pp");
pp.setDoubleValue(1.0);
```

or by using overloaded operators:

```
cPar pp("pp");
pp = 1.0;
```

For reading its value, it is best to use overloaded type cast operators:

```
double d1 = (double)pp;
// or simply:
double d2 = pp;
```

Long integers:

```
pp = 89363L; // or:
pp.setLongValue( 89363L );
```

Character string:

```
pp = "hi there"; // or:
pp.setStringValue( "hi there" );
```

The cPar object makes its own copy of the string, so the original one does not need to be preserved. Short strings (less than ~20 chars) are handled more efficiently because they are stored in the object's memory space (and are not dynamically allocated).

There are several other types cPar can store: such as boolean, void* pointer; cObject* pointer, function with constant args; they will be mentioned in the next section.

For numeric and string types, an input flag can be set. In this case, when the object's value is first used, the parameter value will be searched for in the configuration (ini) file; if it is not found there, the user will be given a chance to enter the value interactively.

Examples:

```
cPar inp("inp");
inp.setPrompt("Enter my value:");
inp.setInput( true );   // make it an input parameter
double a = (double)inp; // the user will be prompted HERE
```

7.6.2 Random number generation through cPar

Setting cPar to call a function with constant arguments can be used to make cPar return random variables of different distributions:

```
cPar rnd("rnd");
rnd.setDoubleValue(intuniform, -10.0, 10.0);// uniform distr.
rnd.setDoubleValue(normal, 100.0, 5.0); // normal distr. (mean,dev)
rnd.setDoubleValue(exponential, 10.0); // exponential distr. (mean)
```

intuniform(), normal() etc. are ordinary C functions taking double arguments and returning double. Each time you read the value of a cPar containing a function like above, the function will be called with the given constant arguments (e.g. normal(100.0,5.0)) and its return value used.

The above functions use number 0 from the several random number generators. To use another generator, use the genk_xxx versions of the random functions:

```
rnd.setDoubleValue(genk\_normal, 3, 100.0, 5.0); // uses generator 3
```

A cPar object can also be set to return a random variable from a distribution collected by a statistical data collection object:

```
cDoubleHistogram hist =....; // the distribution
cPar rnd2("rnd2");
rnd2.setDoubleValue(hist);
```

7.6.3 Storing object and non-object pointers in cPar

cPar can store pointers to OMNeT++ objects. You can use both assignment and the setObjectValue() member function:

```
cQueue *queue = new cQueue("queue"); // just an example
cPar par1, par2;
par1 = (cObject *) queue;
par2.setObjectValue( queue );
```

To get the store pointer back, you can use typecast or the objectValue() member function:

```
cQueue *q1 = (cQueue *)(cObject *)par1;
cQueue *q2 = (cQueue *)par2.objectValue();
```

Whether the cPar object will own the other object or not is controlled by the takeOwnership() member function, just as with container classes. This is documented in detail in the class library reference. By default, cPar will own the object.

cPar can be used to store non-object pointers (for example C structs) or non-OMNeT++ object types in the parameter object. It works very similarly to the above mechanism. An example:

```
double *mem = new double[15];
cPar par1, par2;
par1 = (void *) mem;
par2.setPointerValue( (void *)mem );
...
double *m1 = (double *)(void *)par1;
double *m2 = (double *)par2.pointerValue();
```

Memory management can be specified by cPar's configPointer() member function. It takes three arguments: a pointer to a user-supplied deallocation function, a pointer to a user-supplied duplication function and an item size. If all three are 0 (NULL), no memory management is done, that is, the pointer is treated as a mere pointer. This is the default behaviour. If you supply only the item size (and both function pointers are NULL), cPar will use the delete operator to deallocate the memory area when the cPar object is destructed, and it will use new char[size] followed by a memcpy() to duplicate the memory area whenever the cPar object is duplicated. If you need more sophisticated memory management, you can supply your own deallocation and duplication functions.

An example for simple memory management:

```
double *mem = new double[15];
cPar par;
par.setPointerValue((void *) mem);
par.configPointer(NULL, NULL, 15*sizeof(double));
// -> now if par goes out of scope, it will delete the 15-double array.
```

The configPointer() setting only affects what happens when the cPar is deleted, duplicated or copied, but does *not* apply to assigning new pointers. That is, if *you* assign a new void* to the cPar, you simply overwrite the pointer – the block denoted by the old pointer is *not* deleted. This fact can be used to extract some dynamically allocated block out of the cPar: carrying on the previous example, you would extract the array of 15 doubles from the cPar like this:

```
double *mem2 = (double *)par.pointerValue();
par.setValue( (void *)0 );
// -> now par has nothing to do with the double[15] array
```

However, if you assign some non-pointer value to the cPar, beware: this *will* activate the memory management for the block. If you temporarily use the same cPar object to store other than void* ('P') values, the configPointer() setting is lost.

7.6.4 Reverse Polish expressions

This feature is rarely needed by the user, it is more used internally. A cPar object can also store expressions. In this case, the expression must be given in reversed Polish form. An example:

```
cPar::XElem *expression = new cPar::XElem[5];
expression[0] = &par("count"); // pointer to
module parameter
expression[1] = 1;
expression[2] = '+';
expression[3] = 2;
expression[4] = '/';

cPar expr("expr");
expr.setDoubleValue(expression,5);
```

The cPar object created above contains the $(count+1)/2$ expression where *count* is a module parameter. Each time the cPar is evaluated, it recalculates the expression, using the current value of count. Note the & sign in front of par("count") expression: if it was not there, the

parameter would be taken by value, evaluated once and then the resulting constant would be used.

Another example is a distribution with mean and standard deviation given by module parameters:

```
cPar::XElem *expression = new cPar::XElem[3];
expression[0] = &par("mean");
expression[1] = &par("stddev");
expression[2] = normal; // pointer to the normal(double,double) func.

cPar expr("expr");
expr.setDoubleValue(expression,3);
```

For more information, see the reference and the code NEDC generates for parameter expressions.

7.6.5 Using redirection

A `cPar` object can be set to stand for a value actually stored in another `cPar` object. This is called *indirect* or *redirected* value. When using redirection, every operation on the value (i.e. reading or changing it) will be actually done to the other `cPar` object:

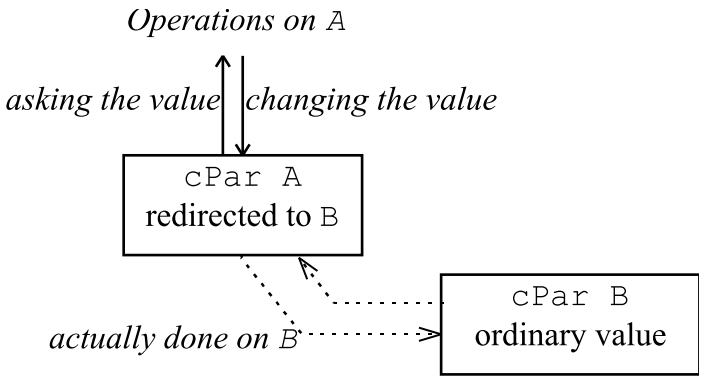

Figure 7.2: cPar redirection

Redirection is how module parameters taken by reference are implemented. The redirection does not include name strings. That is, if you say A->setName("newname") in the above example, A's name will be changed as the name member is not redirected. (This is natural if you consider parameters taken by reference: a parameter should/can have different name than the value it refers to.)

You create a redirection with the `setRedirection()` function:

```
cPar *bb = new cPar("bb"); // background value
bb = 10L;
cPar a("a"); // we'll redirect this object

a.setRedirection(bb); // create redirection
```

Now every operation you do on a's value will be done to bb:

```
long x = a; // returns bb's value, 10L
a = 5;      // bb's value changes to 5
```

The only way to determine whether a is really holding the value or it is redirected to another `cPar` is to use the `isRedirected()` member function which returns a bool, or `redirection()` which returns the pointer to the background object, or NULL if there's no redirection:

```
cPar *redir = a.redirection(); // returns bb's pointer
if (redir != NULL)
  ev << "a is redirected to " << redir->name() << endl;
```

To break the link between the two objects, use the `cancelRedirection()` member function. (No other method will work, including assigning a the value of another `cPar` object.) The `cancelRedirection()` function gives the `(long)0` value to the redirected object (the other will be unaffected). If you want to cancel the indirection but keep the old value, you can do something like this:

```
cPar *value = a.redirection();  // bb's pointer
a.cancelRedirection();          // break the link; value of a is now 0
a = *value;                     // copy the contents of bb into a
```

7.6.6 Type characters

Internally, `cPar` objects identify the types of the stored values by type characters. The type character is returned by the `type()` member function:

```
cPar par = 10L;
char typechar = par.type(); // returns 'L'
```

The full table of type characters is presented in the *Summary* section below.

The `isNumeric()` function tells whether the object stores one of the numeric types, so that e.g. `asDoubleValue()` can be invoked on it.

7.6.7 Summary

The various `cPar` types and the member functions used to manipulate them are summarized in the following table:

Type char	Type name	Member functions	Description

S	string	`setStringValue( const char *);`
		`const char * stringValue();`
		`op const char *();`
		`op=(const char *);`
			string value. Short strings (len<=27) are stored inside `cPar` object, without using heap allocation.

B	boolean	`setBoolValue(bool);`
		`bool boolValue();`
		`op bool();`
		`op=(bool);`
			boolean value. Can also be retrieved from the object as long (0 or 1).

L	long int	`setLongValue(long);`
		`long longValue();`
		`op long();`
		`op=(long);`
			signed long integer value. Can also be retrieved from the object as double.

D	double	`setDoubleValue(double);`
		`double doubleValue();`
		`op double();`
		`op=(double);`
			double-precision floating point value.

F	function	`setDoubleValue(`
		`  MathFunc,`
		`  [double],`
		`  [double],`
		`  [double]);`
		`double doubleValue();`
		`op double();`
			Mathematical function with constant arguments. The function is given by its pointer; it must take 0,1,2 or 3 doubles and return a double. This type is mainly used to generate random numbers: e.g. the function takes mean and standard deviation and returns random variable of a certain distribution.

X	expr.	`setDoubleValue(`
		`  cPar::XElem*,int);`
		`double doubleValue();`
		`op double();`
			Reverse Polish expression. Expression can contain constants, `cPar` objects, refer to other `cPars` (e.g. module parameters), can use many math operators (+-*/^% etc), function calls (function must take 0,1,2 or 3 doubles and return a double). The expression must be given is in an cPar::XElem array (see later).

T	distrib.	`setDoubleValue(`
		`  cStatistic*);`
		`double doubleValue();`
		`op double();`
			random variable generated from a distribution collected by a statistical data collection object (derived from `cStatistic`).

P	void* pointer	`setPointerValue(void*);`
		`void *pointerValue();`
		`op void *();`
		`op=(void *);`
			pointer to a non-`cObject` item (C struct, non-`cObject` object etc.) Memory management can be controlled through the `configPointer()` member function.

O	object pointer	`setObjectValue(cObject*);`
		`cObject *objectValue();`
		`op cObject *();`
		`op=(cObject *);`
			pointer to an object derived from `cObject`. Ownership management is done through `takeOwnership()`.

I	indirect value	`setRedirection(cPar*);`
		`bool isRedirected();`
		`cPar *redirection();`
		`cancelRedirection();`
			value is redirected to another `cPar` object. All value setting and reading operates on the other `cPar`; even the `type()` function will return the type in the other `cPar` (so you'll never get 'I' as the type). This redirection can only be broken with the `cancelRedirection()` member function. Module parameters taken by REF use this mechanism.

7.7 Routing support: cTopology

7.7.1 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.

7.7.2 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 User. (Router and User can be both simple and compound module types.)

```
cTopology topo;
topo.extractByModuleType( "Router", "User", NULL );
```

Any number of module types (up to 32) can be supplied; the list must be terminated by NULL.

Second, you can extract all modules which have a certain parameter:

```
topo.extractByParameter( "ip_address" );
```

You can also specify that the parameter must have a certain value for the module to be included in the graph:

```
cPar yes = "yes";
topo.extractByParameter( "include_in_topo", &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 select_function(cModule *mod, void *)
{
  return mod->parentModule() == simulation.systemModule();
}

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

A cTopology object uses two types: sTopoNode for nodes and sTopoLink for edges. (sTopoLinkIn and sTopoLinkOut are 'aliases' for sTopoLink; 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.nodes(); i++)
{
  sTopoNode *node = topo.node(i);
  ev << "Node i=" << i << " is " << node->module()->fullPath() << endl;
  ev << " It has " << node->outLinks() << " conns to other nodes\n";
  ev << " and " << node->inLinks() << " conns from other nodes\n";

  ev << " Connections to other modules are:\n";
  for (int j=0; j<node->outLinks(); j++)
  {
    sTopoNode *neighbour = node->out(j)->remoteNode();
    cGate *gate = node->out(j)->localGate();
    ev << " " << neighbour->module()->fullPath()
       << " through gate " << gate->fullName() << endl;
  }
}
```

The nodes() member function (1st line) returns the number of nodes in the graph, and node(i) returns a pointer to the *i*th node, an sTopoNode structure.

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

```
sTopoNode *node = topo.nodeFor( module );
cModule *module = node->module();
```

The nodeFor() member function returns a pointer to the graph node for a given module. (If the module is not in the graph, it returns NULL). nodeFor() uses binary search within the cTopology object so it is fast enough.

sTopoNode's other member functions let you determine the connections of this node: inLinks(), outLinks() 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 remoteNode() function returns the other end of the connection, and localGate(), remoteGate(), localGateId() and remoteGateId() return the gate pointers and ids of the gates involved. (Actually, the implementation is a bit tricky here: the same graph edge object sTopoLink is returned either as sTopoLinkIn or as sTopoLinkOut so that "remote" and "local" can be correctly interpreted for edges of both directions.)

7.7.3 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 =...;
sTopoNode *targetnode = topo.nodeFor( targetmodulep );
topo.unweightedSingleShortestPathsTo( targetnode );
```

This performs the Dijkstra algorithm and stores the result in the cTopology object. The result can then be extracted using cTopology and sTopoNode methods. Naturally, each call to unweightedSingleShortestPathsTo() overwrites the results of the previous call.

Walking along the path from our module to the target node:

```
sTopoNode *node = topo.nodeFor( this );

if (node == NULL)
{
  ev < "We (" << fullPath() << ") are not included in the topology.\n";
}
else if (node->paths()==0)
{
  ev << "No path to destination.\n";
}
else
{
  while (node != topo.targetNode())
  {
    ev << "We are in " << node->module()->fullPath() << endl;
    ev << node->distanceToTarget() << " hops to go\n";
    ev << "There are " << node->paths()
       << " equally good directions, taking the first one\n";
    sTopoLinkOut *path = node->path(0);
    ev << "Taking gate " << path->localGate()->fullName()
       << " we arrive in " << path->remoteNode()->module()->fullPath()
       << " on its gate " << path->remoteGate()->fullName() << endl;
    node = path->remoteNode();
  }
}
```

The purpose of the distanceToTarget() member function of a node is self-explanatory. In the unweighted case, it returns the number of hops. The paths() 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 sTopoLinkOut. If the shortest paths were created by the ...SingleShortestPaths() function, paths() 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 targetNode() 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 `enabled()` 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:

```
sTopoNode *thisnode = topo.nodeFor( this );
thisnode->disable();
topo.unweightedSingleShortestPathsTo( targetnode );
thisnode->enable();

for (int j=0; j<thisnode->outLinks(); j++)
{
  sTopoLinkOut *link = thisnode->out(i);
  ev << "Through gate " << link->localGate()->fullName() << " : "
     << 1 + link->remoteNode()->distanceToTarget() << " hops" << endl;
}
```

In the future, other shortest path algorithms will also be implemented:

```
unweightedMultiShortestPathsTo(sTopoNode *target);
weightedSingleShortestPathsTo(sTopoNode *target);
weightedMultiShortestPathsTo(sTopoNode *target);
```

7.8 Statistics and distribution estimation

7.8.1 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 number of samples, mean, standard deviation, minimum and maximum value etc.
- `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 [JC85]. 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: samples(), min(), max(), mean(), stddev(), variance(), sum(), sqrSum() 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.samples();
double smallest = stat.min(),
       largest = stat.max();
double mean = stat.mean(),
       standardDeviation = stat.stddev(),
       variance = stat.variance();
```

7.8.2 Distribution estimation

Initialization and usage

The distribution estimation classes (the histogram classes, 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(num_firstvals, range_ext_factor);
setRangeAutoLower(upper, num_firstvals, range_ext_factor);
setRangeAutoUpper(lower, num, range_ext_factor);

setNumFirstVals(num_firstvals);
```

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

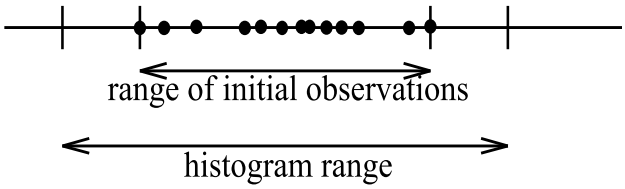

Figure 7.3: Setting up a histogram's range

the chance that further observations fall in one of the cells and not outside the histogram range.

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

The `transformed()` 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 `underflowCell()` and `overflowCell()` member functions.

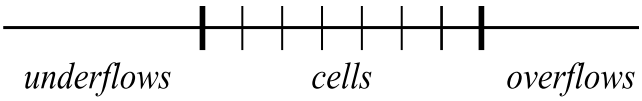

Figure 7.4: Histogram structure after setting up the cells

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. `cells()` returns the number of cells, `basepoint(int k)` returns the kth base point, `cell(int k)` returns the number of observations in cell k, and `cellPDF(int k)` returns the PDF value in the cell (i.e. between `basepoint(k)` and `basepoint(k+1)`). These functions work for all histogram types, plus `cPSquare` and `cKSplit`.

An example:

```
long n = histogram.samples();
for (int i=0; i<histogram.cells(); i++)
{
  double cellWidth = histogram.basepoint(i+1)-histogram.basepoint(i);
  int count = histogram.cell(i);
  double pdf = histogram.cellPDF(i);
  //...
}
```

The `pdf(x)` and `cdf(x)` member functions return the value of the probability density function and the cumulated density function at a given x, respectively.

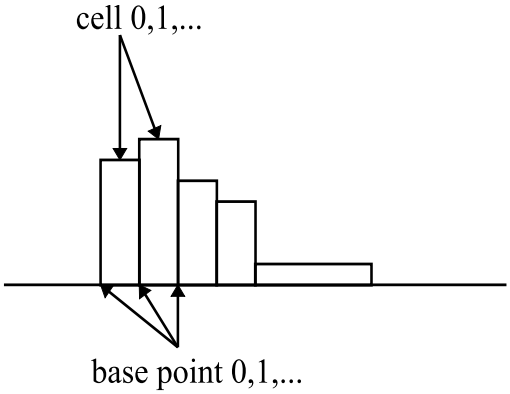

Figure 7.5: base points and cells

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`:

```
cPar rnd_par("rnd_par");
rnd_par.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)rnd_par; // 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 );

FILE *f2 = fopen("histogram.dat","r");}
cDoubleHistogram hist2("Hist-from-file");
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 `num_firstvals` initial observations. One cannot add cell boundaries when the histogram has already been transformed.

7.8.3 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 $n_1, n_2, \cdots n_k$. Each collected observation increments the corresponding observation count. When an observation count n_i reaches a *split threshold*, the cell is split into k smaller, equal-sized cells with observation counts $n_{i,1}, n_{i,2}, \cdots n_{i,k}$ initialized to zero. The n_i observation count is remembered and is called the *mother observation count* to the newly created cells. Further observations may cause cells to be split further (e.g.

$n_{i,1,1}, ...n_{i,1,k}$ etc.), thus creating a k-order tree of observation counts where leaves contain live counters that are actually incremented by new observations, and intermediate nodes contain mother observation counts for their children. If an observation falls outside the histogram range, the range is extended in a natural manner by inserting new level(s) at the top of the tree. The fundamental parameter to the algorithm is the split factor k. Low values of k, $k = 2$ and $k = 3$ are to be considered. In this paper we examine only the $k = 2$ case.

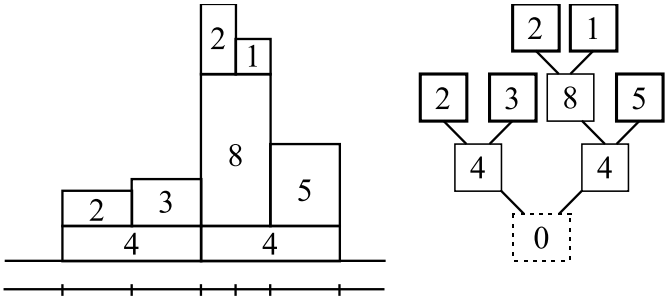

Figure 7.6: Illustration of the k-split algorithm, $k = 2$. The numbers in boxes represent the observation count values

For density estimation, the total number of observations that fell into each cell of the partition has to be determined. For this purpose, mother observations in each internal node of the tree must be distributed among its child cells and propagated up to the leaves.

Let $n_{...,i}$ be the (mother) observation count for a cell, $s_{...,i}$ be the total observation count in a cell $n_{...,i}$ plus the observation counts in all its sub-, sub-sub-, etc. cells), and $m_{...,i}$ the mother observations propagated to the cell. We are interested in the $\tilde{n}_{...,i} = n_{...,i} + m_{...,i}$ estimated amount of observations in the tree nodes, especially in the leaves. In other words, if we have $\tilde{n}_{...,i}$ estimated observation amount in a cell, how to divide it to obtain $m_{...,i,1}, m_{...,i,2} \cdots m_{...,i,k}$ that can be propagated to child cells. Naturally, $m_{...,i,1} + m_{...,i,2} + \cdots + m_{...,i,k} = \tilde{n}_{...,i}$.

Two natural distribution methods are even distribution (when $m_{...,i,1} = m_{...,i,2} = \cdots = m_{...,i,k}$) and proportional distribution (when $m_{...,i,1} : m_{...,i,2} : \cdots : m_{...,i,k} = s_{...,i,1} : s_{...,i,2} : \cdots : s_{...,i,k}$). Even distribution is optimal when the $s_{...,i,j}$ values are very small, and proportional distribution is good when the $s_{...,i,j}$ values are large compared to $m_{...,i,j}$. In practice, a linear combination of them seems appropriate, where $\lambda = 0$ means even and $\lambda = 1$ means proportional distribution:

$$m_{...,i,j} = (1 - \lambda)\frac{\tilde{n}_{...,i}}{k} + \lambda\tilde{n}_{...,i}\frac{s_{...,i,j}}{s_{...,i}}, \lambda \in [0, 1] \tag{7.1}$$

Note that while $n_{...,i}$ are integers, $m_{...,i}$ and thus $\tilde{n}_{...,i}$ are typically real numbers. The histogram estimate calculated from *k*-split is not exact, because the frequency counts calculated in the above manner contain a degree of estimation themselves. This introduces a certain *cell division error*; the λ parameter should be selected so that it minimizes that error. It has been shown that the cell division error can be reduced to a more-than-acceptable small value. Strictly speaking, the *k*-split algorithm is semi-online, because its needs some observations to set up the initial histogram range. However, 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 enough 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

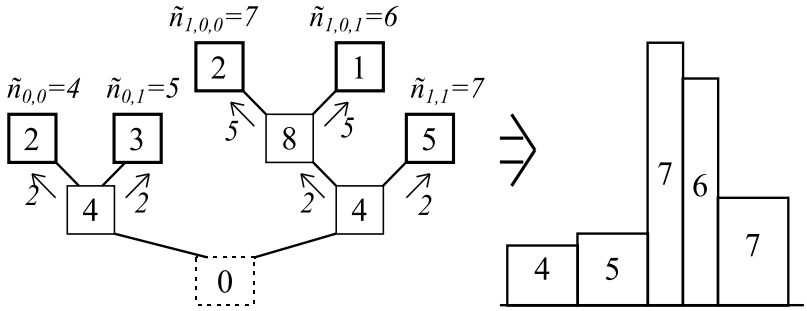

Figure 7.7: Density estimation from the k-split cell tree. We assume $\lambda = 0$, i.e. we distribute mother observations evenly.

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 treeDepth();
int treeDepth(sGrid& grid);

double realCellValue(sGrid& grid, int cell);
void printGrids();

sGrid& grid(int k);
sGrid& rootGrid();

struct sGrid
{
  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
};
```

7.8.4 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 have 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 *transientDetectionObject();
cAccuracyDetection *accuracyDetectionObject();
```

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);
```

7.9 Recording simulation results

7.9.1 Output vectors: cOutVector

Objects of type `cOutVector` are responsible for writing time series data (referred to as *output vectors*) to a file. The `record()` member is used to output a value (or a value pair) with a timestamp.

It can be used like this:

```
cOutVector resp_v("response time");

while (...)
{
  double response_time;
  //...
  resp_v.record( response_time );
  //...
}
```

All `cOutVector` objects write to the same, common file. The file is textual; each `record()` call generates a line in the file. The output file can be processed using Plove, but otherwise its simple format allows it to be easily processed with `sed`, `awk`, `grep` and the like, and it can be imported by spreadsheet programs. The file format is described later in this manual (in the section about simulation execution).

You can disable the output vector or specify a simulation time interval for recording either from the ini file or directly from program code:

```
cOutVector v("v");
simtime_t t =...;

v.enable();
v.disable();
v.setStartTime( t );
v.setStopTime( t+100.0 );
```

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.

7.9.2 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 outputs 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 `recordScalar()` member function. It is overloaded, you can use it to write doubles and strings (const char *):

```
double avg_throughput = total_bits / simTime();
recordScalar("Average throughput", avg_throughput);
```

You can record whole statistics objects by calling `recordStats()`:

```
cStdDev *eedstats = new cStdDev;
...
recordStats("End-to-end Statistics", eedstats);
```

Calls to `recordScalar()` and `recordStats()` are usually placed in the redefined `finish()` member function of a simple module.

The above calls write into the (textual) output scalar file. The output scalar file is preserved across simulation runs (unlike the output vector file is, scalar files are not deleted at the beginning of each run). Data are always appended at the end of the file, and output from different simulation runs are separated by special lines.

7.10 Tracing and debugging aids

7.10.1 Displaying information about module activity

You can have simple modules print textual output for debugging purposes.

The global object called `ev` represents the user interface of the simulation program. You can send data to `ev` using the C++-style I/O operator (<<).

```
ev << "started\n";
ev << "about to send message #" <<  i << endl;
ev << "queue full, discarding packet\n";
```

The more traditional-looking but functionally equivalent `ev.printf()` form also exists.

```
ev.printf("%d packets dropped out of %d\n", drops, total);
```

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 the ini file so that it doesn't slow down simulation when it is not needed.) In windowing user interfaces (Tkenv), each simple module can have a separate text output window.

The above means that you should *not* use `printf()`, `cout` « and the like because with Tkenv, their output would appear in the terminal window behind the graphical window of the simulation application.

7.10.2 Watches

You may want some of your int, long, double, char, etc. variables to be inspect-able in Tkenv and to be output into the snapshot file. In this case, you can create `cWatch` objects for them with the `WATCH()` macro:

```
int i; WATCH(i);
char c; WATCH(c);
```

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 expands to a dynamically created `cWatch` object. The object remembers the address and type of your variable. The macro expands to something like:

```
new cWatch("i",i);
```

You can also make a `WATCH` for pointers of type `char*` or `cObject*`, but this may cause a segmentation fault if the pointer does not point to a valid location when Tkenv or `snapshot()` wants to use it.

You can also set watches for variables that are members of the module class or for structure fields:

```
WATCH( lapbconfig.timeout );
```

Placement of WATCHes

Be careful not to execute a `WATCH()` statement more than once, as each call would create a new `cWatch` object! If you use `activity()`, the best place for WATCHes is the top of the `activity()` function. If you use `handleMessage()`, place the `WATCH()` statement into `initialize()`. `WATCH()` creates a dynamic `cWatch` object, and we do not want to create a new object each time `handleMessage()` is called.

7.10.3 Snapshots

The `snapshot()` function outputs textual information about all or selected objects of the simulation (including the objects created in module functions by the user) into the snapshot file.

```
bool snapshot(cObject *obj = &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:

```
================================================
|| SNAPSHOT ||
================================================
| Of object:    'simulation'
| Label:        'three-station token ring'
| Sim. time:    0.0576872457 ( 57ms)
| Network:      'token'
| Run no.       1
| Started at:   Mar 13, 1997, 14:23:38
| Time:         Mar 13, 1997, 14:27:10
| Elapsed:      5 sec
| Initiated by: operator
================================================


(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

(cCompoundModule) '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

(cModulePar) 'token.num_stations' begin
  Type: L
  Value: 3
end
```

```
[...token.num_messages omitted...]

(cModulePar) 'token.ia_time' begin
  Type:  F
  Value: truncnormal(0.005,0.003)
end

[...rest of parameters & gates stuff deleted from here...]

(cCompoundModule) 'token.comp[0]' begin
  parameters    (cArray) (empty)
  gates         (cArray) (n=2)
  mac           (TokenRingMAC,#3)
  gen           (Generator,#4)
  sink          (Sink,#5)
end

(cArray) 'token.comp[0].parameters' begin
end

(cArray) 'token.comp[0].gates' begin
  in            (cGate)  <-- comp[2].out
  out           (cGate)  --> D --> comp[1].in
end

(cGate) 'token.comp[0].in' begin
  type:  input
  inside connection:  token.comp[0].mac.phy_in
  outside connection: token.comp[2].out
  delay: -
  error: -
  data rate: -
end

(cGate) 'token.comp[0].out' begin
  type: output
  inside connection: token.comp[0].mac.phy_out
  outside connection: token.comp[1].in
  delay: (cPar) 1e-06 (D)
  error: -
  data rate: -
end

(TokenRingMAC) 'token.comp[0].mac' begin
  parameters    (cArray) (n=2)
  gates         (cArray) (n=4)
  local-objects (cHead)
  class-data-members (cHead)
end

[...comp[0].mac parameters stuff deleted from here...]

(cArray) 'token.comp[0].mac.gates' begin
  phy_in        (cGate)  <-- <parent>.in
```

```
  from_gen      (cGate)  <-- gen.out
  phy_out       (cGate)  --> <parent>.out
  to_sink       (cGate)  --> sink.in
end

[...detailed gate list deleted from here...]

(cHead) 'token.comp[0].mac.local-objects' begin
  sendqueue-length (cOutVector) (single)
  send-queue   (cQueue) (n=11)
end

(cOutVector) 'token.comp[0].mac.local-objects.sendqueue-length' begin
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

(cArray) 'token.comp[0].mac.local-objects.send-queue.0-->1.par-vector' begin
  dest          (cPar) 1 (L)
  source        (cPar) 0 (L)
  gentime       (cPar) 0.0158106 (D)
end

[...message parameters and the other messages' stuff deleted...]

(cHead) 'token.comp[0].mac.class-data-members' begin
end
```

```
[...comp[0].gen and comp[0].sink stuff deleted from here...]
[...whole comp[1] and comp[2] stuff deleted from here...]

(cMessageHeap) 'simulation.message-queue' begin
  1-->0         (cMessage) Tarr=0.0576872457 ( 57ms) Src=#8 Dest=#7
                (cMessage) Tarr=0.0577201630 ( 57ms) Mod=#8 (selfmsg)
                (cMessage) Tarr=0.0585677054 ( 58ms) Mod=#4 (selfmsg)
                (cMessage) Tarr=0.0594939072 ( 59ms) Mod=#12 (selfmsg)
                (cMessage) Tarr=0.0601010000 ( 60ms) Mod=#7 (selfmsg)
  1-->2         (cMessage) Tarr=0.0601020000 ( 60ms) Src=#11 Dest=#13
end

[...detailed list of message queue contents deleted from here...]
```

7.10.4 Breakpoints

With activity() only! In those user interfaces which support debugging, breakpoints stop execution and the state of the simulation can be examined.

You can set a breakpoint inserting a `breakpoint()` call into the source:

```
for(;;)
{
  cMessage *msg = receive();
  breakpoint("before-processing");
  breakpoint("before-send");
  send( reply_msg, "out" );
  //..
}
```

In user interfaces that do not support debugging, `breakpoint()` calls are simply ignored.

7.10.5 Disabling warnings

Some container classes and functions suspend the simulation and issue warning messages in potentially bogus/dangerous situations, for example when an object is not found and NULL pointer/reference is about to be returned. Very often this is useful, but sometimes it is more trouble. You can turn warnings on/off from the ini file (warnings=yes/no).

It is a good practice to leave warnings enabled, and temporarily disable warnings in places where OMNeT++ would normally issue warnings but you know the code is correct. This is done in the following way:

```
bool w = simulation.warnings();
simulation.setWarnings( false );
...
... // critical code
...
simulation.setWarnings( w );
```

7.10.6 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 `stackUsage()` call which tells you how much stack the module actually uses. It is most conveniently called from `finish()`:

```
void FooModule::finish()
{
  ev << stackUsage() <<  "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. [2]

`stackUsage()`also works by checking the existence of predefined byte patterns in the stack area, so it is also subject to the above effect with local variables.

7.11 Changing the network graphics at run-time

7.11.1 Setting display strings

Sometimes it is useful to change the appearance or position of some components in the network graphics, such as the color of the modules, color/width of connection arrows, position of a submodule, etc.

The appearance of nodes and connections is determined by the display strings. Display strings (e.g. `"p=100,10;i=pc"`) are initially taken from the NED description. You can change the display string of a module or connection arrow at run-time by calling methods named `setDisplayString()`. The `cDisplayStringParser` class (discussed in the following sections) might be useful for manipulating the display string.

Setting the module's appearance when it is displayed as a component within a compound module:

```
setDisplayString("p=100,100;b=60,30,rect;o=red,black,3", true);
```

Setting appearance of a compound module when it's displayed as a bounding box for its submodules:

```
parentModule()->setDisplayStringAsParent("p=100.....", true);
```

The display string of a connection arrow is stored in its source gate, so you'll need to write something like this:

[2] The actual value is dependent on the operating system, e.g. SUN Solaris needs more space.

```
gate("out")->setDisplayString("o=yellow,3");
```

The `setDisplayString()` methods additionally take a bool argument called `immediate`. It specifies whether the display string change should take effect immediately, or only after processing the current event (the default is *immediate=true*). If several display string changes are going to be done within one event, then *immediate=false* is useful because it reduces the number of necessary redraws. *Immediate=false* also uses less stack. But its drawback is that a `setDisplayString()` followed by a `send()` would actually be displayed in reverse order (message animation first), because message animations are performed immediately (actually within the `send()` call).

7.11.2 The cDisplayStringParser class

The `cDisplayStringParser` utility class lets you parse and manipulate display strings.

As far as `cDisplayStringParser` 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:

```
cDisplayStringParser dispstr("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.getString();  // result: "x=joe,,jim;a=1,2;p=beta,,3"
```

7.12 Deriving new classes

7.12.1 cObject 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 `cObject` or not. Note that we're *not* saying you should always subclass from `cObject`. Both solutions have advantages and disadvantages which you have to consider individually for each class.

`cObject` already carries (or provides framework for) significant functionality that are either important for your particular purpose or not. Subclassing `cObject` generally means you have more code to write (as you *have to* redefine certain virtual functions and adopt to conventions) and your class will be a bit more heavy-weight. In turn, it will integrate into OMNeT++ better, for example it will be more visible in Tkenv and can be automatically garbage-collected (this will be discussed later). 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 `cObject`. [3]

The most significant features `cObject` has is the name string (which has to be stored somewhere, so it has its overhead) and ownership management (see section 7.13) 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 `cObject`. On the other hand, if you want 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 `cObject`. If your class fits neither category, you'll need to see if `cObject` brings any benefit for you, and decide accordingly.

7.12.2 cObject virtual methods

Most classes in the simulation class library are descendants of `cObject`. If you want to derive a new class from `cObject` or a `cObject` 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. Do not be embarrassed at the length of the list: most functions are not absolutely necessary to implement. For example, you do not need to redefine `forEach()` 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 (`name()`) of the other object it receives, then call the assignment operator (see below).
- *Destructor*. Any good-tempered class has a destructor.
- *Duplication function,* `cObject *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.
- *Assigment 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.

The following function **should** be implemented if your class contains via pointers or as data member) other object subclassed from `cObject`.

- *Iteration function,* `void forEach(ForeachFunc f)`. The implementation should call the function passed for each object it contains via pointer or as data member; see the API Reference on `cObject` on how to implement `foreach()`. `foreach()` is used by Tkenv and `snapshot()` to navigate, search or display the object tree.

The following methods are **recommended** to implement:

[3] For simplicity, in the these sections "OMNeT++ object" should be understood as "object of a class subclassed from `cObject`"

- *Object info,* `void info(char *)`. The `info()` function should print a one-line info about object contents into the given buffer – usually the class name, the object name, important state variables, etc. This is used when Tkenv displays list of objects (in the object tree or in listboxes). The length of the info should not exceed 500 chars.
- *Detailed object info,* `void writeContents(ostream&)`. It should write a detailed multi-line report about the object contents into the stream provided. This is currently only used by `snapshot()`.

7.12.3 Class registration

You should also use the `Register_Class()` macro to register the new class. It is used by the `createOne()` function.

7.12.4 Details

We'll go through the details using an example. We create a new class `NewClass`, redefine all above mentioned `cObject` member functions, and explain the conventions, rules and tips associated with them. To demonstrate as much as possible, the class will contain an `int` data member, dynamically allocated non-`cObject` data (an array of `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 cObject
{
  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 cObject *dup() const;
    NewClass& operator=(const NewClass& other);

    virtual void foreach(ForeachFunc f);

    virtual void info(char *buf);
    virtual void writeContents(ostream& os);
    ...
};
```

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) : cObject(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. `cObject`-based data members should have their owners explicitly set to NULL.

```
NewClass::NewClass(const NewClass& other) : cObject(name())
{
    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.name())` 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.

`cObject`-based data members should have their owners explicitly set to NULL.

```
NewClass::~NewClass()
{
    delete [] array;
    if (msg->owner()==this)
        delete msg;
}
```

The destructor should delete all data structures the object allocated. `cObject`-based objects should *only* be deleted if they are owned by the object – details will be covered in section 7.13.

```
cObject *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;
    cObject::operator=(other);

    data = other.data;

    for (int i=0; i<10; i++)
        array[i] = other.array[i];

    queue = other.queue;
    queue.setName(other.queue.name());

    if (msg && msg->owner()==this)
        delete msg;
    if (other.msg && other.msg->owner()==const_cast<cMessage*>(&other))
        take(msg = (cMessage *)other.msg->dup());
    else
        msg = other.msg;
    return *this;
}
```

Complexity associated with copying and duplicating the object is centralized in the assignment operator, so it is usually the one that requires the most work from you of all methods required by cObject.

If you do not want to implement object copying and duplication, you should implement the assigment 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 7.13.

```
void NewClass::forEach(ForeachFunc f)
{
    if (f(this,true))
    {
        queue->forEach(f);
        if (msg)
            msg->forEach(f);
```

```
    }
    f(this,false);
}
```

The `foreach()` function should be called for each OMNeT++ member of the class. See the API Reference for more information of `foreach()`.

```
void NewClass::info(char *buf)
{
    cObject::info(buf);
    sprintf(buf+strlen(buf), " data=%d, array[0]=%g, %s",
            data, array[0], (msg ? "has msg" : "no msg"));
}
```

Here you should produce a concise, one-line info about the object. You should try not to exceed 40-80 characters, since the string will be shown in tooltips and listboxes. The length of the buffer is 500 bytes, so in any case you should not exceed that length. You can make use of `cObject`'s `info()` method that produces the class name and the object name.

```
void NewClass::writeContents(ostream& os)
{
    os << " data: " << data << endl;
    os << " array: "
    for (int i=0; i<10; i++)
        os << array[i] << " ";
    os << endl;
}
```

`writecontents()` is expected to write values of all data members to the stream. You do not need to include anything about contained `cObject`-based objects, because they will be included via `foreach()`.

See the virtual functions of `cObject` in the class library reference for more information. The sources of the Sim library (`include/`, `src/sim/`) can serve as further examples.

7.13 Object ownership management

OMNeT++ has a built-in ownership management mechanism which is used for garbage collection, sanity checks, and as part of the infrastructure supporting Tkenv inspectors. It usually works transparently, but it is useful to know what it does exactly so that it doesn't interfere with the cleanup code and destructors you write.

If you plan to program small simple modules only, you can probably safely skip this section. But if your simple module code is getting more complex and you're getting memory leaks or seemingly unexplicable segmentation faults because of double deletion of objects, it is probably time to read the following discussion.

7.13.1 Ownership tree

Any `cObject`-based object can be both *owner* of other objects and can at the same time be *owned* by another object. For example, a message object (`cMessage`) may reside in a queue (`cQueue`) and be owned by that queue, while it may own attached `cPar` objects or another message (added to it via `encapsulate()`).

From an object you can navigate to its owner by calling the `owner()` method, defined in `cObject`. The other direction, enumerating the objects owned can be done via `foreach()` that loops through all contained objects and checking the owner of each object.

7.13.2 Purpose

The purpose of maintaining the ownership tree is threefold:

- to provide a certain degree of garbage collection (that is, automatic deletion of objects that are no longer needed)
- to prevent a certain types of programming errors, namely, those associated with wrong ownership handling.
- it provides some "reflection" (in the Java sense), which enables Tkenv to display which objects are present (and where) in the simulation, to find "lost" (leaked) objects, etc.

Some examples of programming errors that can be caught by the ownership facility:

- attempts to send a message while it's still in a queue, encapsulated in another message, etc.
- attempts to send/schedule a message while it's 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)

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.

7.13.3 Objects are deleted by their owners

The concept of `ownership` is that *the owner has the exclusive right and duty to delete the objects it owns*.

As an example, this means if you delete a message, its encapsulated message (see `encapsulate()` method) and attached `cPar` objects are also deleted. If you delete a queue, all messages it contains and also owns will also be deleted. [4]

If you create a new class, you should implement it so that it deletes the *owned* objects in the destructor – that is, you have to check `owner()` of each object before you delete it.

7.13.4 Ownership is managed transparently

Automatic transfer of ownership

Ownership is usually established and managed automatically. It is not hard to guess that objects (i.e. messages) inserted into a `cQueue` or a `cArray` will be owned by that object (by

[4] Note that it's not necessary for a container object like a queue to actually own all inserted objects. This behavior can be controlled via the *takeownership* flag, as explained later.

default – this can be changed, as described later). Messages encapsulated into other messages (by `cMessages`'s `encapsulate()` method), and `cPar`'s added to a message will also become owned by the message object (again, this can be changed), and they are deallocated automatically when the message object is deleted.

The *local objects* list

But objects which, not being stored in another object, appear not to have owners usually have one as well. If you just create a message object inside a simple module (e.g. from `activity()`, `handleMessage()` or any function called from them), it will be *owned by simple module*, or more precisely, by its *"local objects list"* (a member of `cSimpleModule`).

So the following line:

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

actually creates the message object and automatically adds it to the module's local objects list. [5]

The local objects list also plays a role when an object is removed from a container object (e.g. when a message is removed from a queue). In that case, the container object "drops" the ownership of the object, and the object will "join" its default owner, the local objects list of the simple module (again, to the *currently active* simple module's list). Thus, an innocent-looking

```
cMessage *msg = queue.pop();
```

statement actually involves a transfer of ownership of the message object from the queue to the simple module. The same thing happens when a message is decapsulated from another message, when `cPar`'s are removed from a `cArray`, and in many more cases.

Sanity checks

The `send()` and `scheduleAt()` functions make use of the ownership mechanism to do some sanity check: the message being sent/scheduled *must* be owned by module's local objects list. If it is not, then it's an error, because then the message is probably with another module (i.e. already sent), or currently scheduled, or inside a queue, a message or some other object – in either case, you do not have any authority to send it. When you get this error message (`"not owner of object"`), you might feel tempted to forcibly take ownership of the message object by means of `setOwner()`. Note that it would be entirely wrong, and would probably lead to crash further on in your program. *Do not use* `setOwner()`! Instead, you need to carefully examine who has the ownership of the message, why's that, and then probably you'll need to fix the logic somewhere in your program.

The *class members* list

For completeness, it should also be mentioned that class members of a simple module are collected on a *"class members list"*. The reason for the existence of this list is not so much garbage collection or sanity checks, but rather assisting Tkenv in displaying the class members list in simple module inspectors.

[5] More precisely: to the *currently executing* module's local object list, because that's the best guess a `cMessage` constructor can do.

7.13.5 Garbage collection

How it's done

The local objects list is also the reason why you rarely need to put delete statements in your simple module destructors.

When you restart the simulation in Tkenv or begin a new run in Cmdenv, OMNeT++ has to clean up the previously running simulation. This involves (a) deleting all modules in the network, and (b) deleting all messages in the Future Events Set. Modules (both simple and compound) can also be dynamically deleted during simulation (deleting a compound module just consists of recursively deleting all submodules). At that time, one expects all dynamically allocated objects to be properly destructed and memory released, which is not trivial since the simulation kernel does not know what objects have been created by simple modules. Here's how it is done in the simulation kernel.

When a simple module gets deleted, the local objects list is also deleted in addition to the module's gates and parameters. This means that all objects on the module's local objects list (i.e. objects you allocated and need to be garbage collected) will also be deleted, and this is exactly what we need as garbage collection.

The result is that as long as you only have dynamically allocated memory as (or within) `cObject`-based objects, you don't have to worry about writing module destructors: everything is taken care of automatically.

Garbage collection and your module destructors

Note that this garbage collection can nicely co-exist with module destructors you write. If you delete an object explicitly, it is redundant but does no harm: its destructor will also remove it from the owner's list (which might be the module's local object list), so double deletion will not occur.

```
class MyModule : public cSimpleModule
{
    ...
    cMessage *timeoutmsg;
    ...
};

MyModule::~MyModule()
{
    delete timeoutmsg;   // redundant but does no harm
}
```

Other allocated memory (e.g. C++ arrays of integers, doubles, structs or pointers) or objects which have nothing to do with `cObject` (e.g. STL objects or your non-`cObject` rooted classes) are invisible to the ownership mechanism discussed here, and must be deleted in the destructor in the conventional way.

```
class MyModule : public cSimpleModule
{
    ...
    double *distances; // array allocated via new double []
    ...
};
```

```
MyModule::~MyModule()
{
    delete [] distances; // OMNeT++ knows nothing about this vector,
                         // so we need to clean up it manually
}
```

It is similar when you have arrays of `cMessage` pointers (or in general any other non-OMNeT++ data structure which holds pointers to `cObject`-rooted objects). Then it is enough if you delete the array or the data structure, the objects will be cleaned up via the garbage collection mechanism.

```
class MyModule : public cSimpleModule
{
    ...
    cMessage **events; // array allocated via new cMessage *[]
    ...
};

MyModule::~MyModule()
{
    delete [] events; // we need to delete only the pointer array itself,
                      // deleting the message objects can be left to
                      // the garbage collection mechanism
}
```

In any case, remember *not* to put any destructor-like code inside the module's `finish()` function. The main reason is that whenever your simulation stops with an error (or you just stop it within Tkenv), the `finish()` functions will not be called and thus, memory will be leaked.

Can it crash?

A potential crash scenario is when the object ownership mechanism deletes objects before your code does, and *your code, not aware of the ownership mechanism and not knowing that the objects have already been deleted, tries to delete them again*. Note that *this cannot happen* as long as objects stay within the module, because the garbage collection mechanism is embedded deeply in the base class of your simple module, thus it is guaranteed by C++ language rules to take place after all your destructor-related code (your simple module class's destructor and the destructors of data members you added to the simple module class) have executed.

However, if some of your objects have been sent to other modules (e.g. inside a message) while their ownership stayed with the original module (which is a situation one should not allow to happen), the above order of destruction is not guaranteed and crash is possible. To produce the above crash, however, one must work hard to add a nonstandard way of storing objects in a message. This situation will be discussed later in more detail, after we've discussed how containers like `cQueue` and `cArray` work.

Garbage collection of activity() simple modules

Another interesting aspect is what happens when an `activity()` simple module is deleted. Objects that were local variables of `activity()` are just left on the coroutine stack. They

themselves need not (and must not) be deleted using the `delete` operator, but they need to be properly destructed (their destructors called) so that the memory *they* allocated can be released. As of OMNeT++ version 2.3, this is done by actually calling the method named `discard()` in the `cObject` destructor instead of the directly the `delete` operator. `discard()` invokes either the `delete` operator (if the object was allocated dynamically) or directly the object's destructor (if the object was a local variable in `activity()` or a function called from `activity()`). In future releases, the implementation might be changed to rely on C++ exceptions (stack unwinding) for proper cleanup.

7.13.6 What cQueue and cArray do

How can the ownership mechanism operate transparently? It is 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 `cObject`-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 `owner()` 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. The flag is part of `cObject` so that every container object can make use of it, and can be get/set via the `takeOwnership()` and `setTakeOwnership()` methods.

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(cObject *obj)
{
    // insert into queue data structure (linked list)
    ...

    // take ownership if needed
    if (takeOwnership())
        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's not needed any more, you can delete it.

The *release ownership* phrase requires further explanation. When you remove and object from a queue or array, the ownership is expected to be transferred to the simple module's local objects list. This is acomplished by the drop() function, which transfers the ownership to the object's default owner. defaultOwner() is a virtual method returning cObject* defined in cObject, 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: [6]

```
cObject *cQueue::remove(cObject *obj)
{
    // remove object from queue data structure (linked list)
    ...

    // release ownership if needed
    if (obj->owner()==this)
        drop(obj);

    return obj;
}
```

Destructor

The destructor should delete all data structures the object allocated. From the contained objects, only the owned ones are deleted – that is, where obj->owner()==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 just 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 at 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 assigment, while dup() is implemented as new cArray(*this)).

7.13.7 Change of implementation

In the current release (version 2.3), the data structure used to maintain the ownership tree is in cObject. The ownership principle is also enforced in cObject, so it is the cObject

[6] Actual code in src/sim is structured somewhat differently, but the meaning is the same.

destructor that deletes all owned objects. [7]

This behaviour will probably be changed in the next major release, and every class will be made responsible for deleting its own owned objects. This will be closer to the usual C++ practice and will make the OMNeT++ simulation library easier to understand. Also, it will be more efficient with both memory and execution time, without losing significant functionality.

The change will be transparent to simulations, unless you implemented a container class which relies on `cObject`'s destructor to destroy owned objects.

As of the 2.3 release, `cObject` contains 4 pointers. *ownerp* points to the owner, *firstchildp* points to the first owned object, while *prevp*, *nextp* are used to build a doubly linked list of objects held by the same owner. These pointers are private data members, they cannot be accessed directly, only via certain member functions. Changing the owner of an object (`setOwner()` method in `cObject`) involves about 8-9 pointer assignments (i.e., *ownerp*, *prevp*, *nextp* and *firstchildp* in the object, in its owner and siblings).

This data structure is likely to change: *firstchildp*, *prevp* and *nextp* will be removed from `cObject`, and only *ownerp* will remain. The `setOwner()` method will probably be removed entirely.

7.14 Tips for speeding up the simulation

Here are a few tips that can help you make the simulation faster:

- Use message subclassing instead of adding `cPar`'s to messages.
- Try to minimize message creations and deletions. Reuse messages if possible.
- Turn off the display of screen messages when you run the simulation. You can do this in the ini file. Alternatively, you can place #ifdefs around your `ev«` and calls and turn off the define when compiling the simulation for speed.
- Store the module parameters in local variables to avoid calling `cPar` member functions every time.

[7] This is also the reason why there are currently so few `delete` calls in the simulation kernel sources: container classes like `cArray` or `cQueue` leave the task of deleting the contained objects they own to the `cObject` destructor.

Chapter 8

Building Simulation Programs

8.1 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` suffix.
- Messsage definitions, in files with `.msg` suffix.
- Simple modules implementations and other C++ code, in `.cc` files (or `.cpp`, on Windows)

To build an executable simulation program, first you need to translate the NED files and the message files into C++, using the NED compiler (`nedc`) and 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 on Unix/Linux, and `.obj` on Windows), and all object files need to be linked with the necessary libraries to get an executable.

File names for libraries differ for Unix/Linux and for Windows, and it's also different for static and shared libraries. Suppose you have a library called Tkenv. On a Unix/Linux system, the file name for the static library would be something like `libtkenv.a` (or `libtkenv.a.`*<version>*), and the shared library would be called `libtkenv.so` (or `libtkenv.so.`*<version>*). The Windows version of the static library would be `tkenv.lib`, and the DLL (which is the Windows equivalent of shared libraries) would be a file named `tkenv.dll`.

You'll need to link with the following libraries:

- The simulation kernel and class library, called *sim_std* (file `libsim_std.a`, `sim_std.lib`, etc).
- User interfaces. The common part of all user interfaces is the *envir* library (file `libenvir.a`, etc), and the specific user interfaces are *tkenv* and *cmdenv* (`libtkenv.a`, `libcmdenv.a`, etc). You have to link with *envir*, plus either *tkenv* or *cmdenv*.

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.

This section discusses how to use the simulation system on the following platforms:

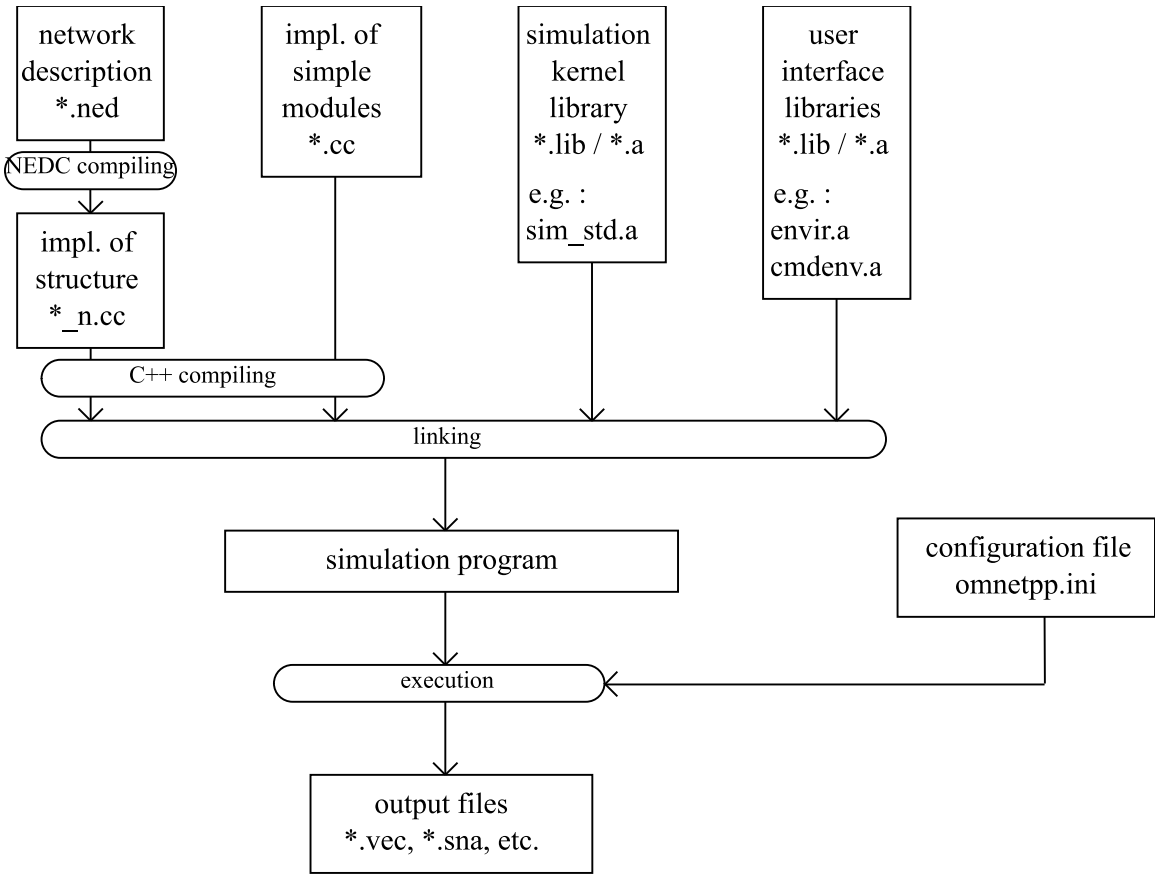

Figure 8.1: Building and running simulation

- Unix with gcc (also Windows with Cygwin or MinGW)
- MSVC 6.0 on Windows

8.2 Using Unix and gcc

This section applies to using OMNeT++ on Linux, Solaris, FreeBSD and other Unix derivatives, and also more or less to Cygwin and MinGW on Windows.

Here in the manual we can give you a rough overview only. The `doc/` directory of your OMNeT++ installation contains `Readme.<`*platform*`>` files that provide up-to-date, more detailed and more precise instructions.

8.2.1 Installation

The installation process depends on what distribution you take (source, precompiled RPM, etc.) and it may change from release to release, so it is better to refer to the readme files. If you compile from source, you can expect the usual GNU procedure: `./configure` followed by `make`.

8.2.2 Building simulation models

The `opp_makemake` script can automatically generate the `Makefile` for your simulation program, based on the source files in the current directory. (It can also handle large models which are spread across several directories; this is covered later in this section.)

`opp_makemake` has several options, with the `-h` option it displays a summary.

```
% opp_makemake -h
```

Once you have the source files (`*.ned`, `*.msg`, `*.cc`, `*.h`) in a directory, `cd` there then type:

```
% opp_makemake
```

This will create a file named `Makefile`. Thus if you simply type `make`, your simulation program should build. The name of the executable will be the same as the name of the directory containing the files.

The freshly generated `Makefile` doesn't contain dependencies, it is advisable to add them by typing `make depend`. The warnings during the dependency generation process can be safely ignored.

In addition to the simulation executable, the `Makefile` contains other targets, too. Here is a list of important ones:

Target	Action
	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

neddoc.html	Generates documentation for all NED files. The resulting file is named `neddoc.html`
htmldoc	Generates code documentation using doxygen. The documentation will be placed into the directory `htmldoc`
doc	Convenience target that calls neddoc.html and htmldoc
re-makemake	Regenerates the `Makefile` using `opp_makemake` (this is useful if e.g. after upgrading OMNeT++, if `opp_makemake` has changed)
re-makemake-m	Similar to `make re-makemake`, but it regenerates the `Makefile.in` instead

If you already had a `Makefile` in that directory, `opp_makemake` will refuse overwriting it. You can force overwriting the old `Makefile` with the -f option:

```
% opp_makemake -f
```

If you have problems, check the path definitions (locations of include files and libraries etc.) in the configure script and correct them if necessary. Then re-run configure to commit the changes to all makefiles, the `opp_makemake` script etc.

You can specify the user interface (Cmdenv/Tkenv) with the -u option (with no -u, Tkenv is the default):

```
% opp_makemake -u Tkenv
```

Or:

```
% opp_makemake -u Cmdenv
```

The name of the output file is set with the -o option (the default is the name of the directory):

```
% opp_makemake -o fddi-net
```

If some of your source files are generated from other files (for example, you use machine-generated NED files), write your make rules into a file called `makefrag`. 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 `Makefile`.

If you want better portability for your models, you can generate `Makefile.in` instead of `Makefile` with `opp_makemake`'s -m option. You can then use `autoconf`-like configure scripts to generate the `Makefile`.

8.2.3 Multi-directory models

In the case of a large project, your source files may be spread across several directories. You have to decide whether you want to use static linking, shared or run-time loaded (shared) libraries. Here we discuss static linking.

In each subdirectory (say `app/` and `routing/`), run

```
opp_makemake -n
```

The -n option means no linking is necessary, only compiling has to be done.

In your toplevel source directory, run

```
opp_makemake app/ routing/
```

This results in recursive makefiles: when you build the simulation, make will descend into `app/` and `routing/`, run make in both, then it will link an executable with the object files in the two directories.

You may need to use the -I option if you include files from other directories. The -I option is for both C++ and NED files. In our example, you could run

```
opp_makemake -n -I../routing
```

in the `app/` directory, and vice versa.

If you're willing to play with shared and run-time loaded libraries, several `opp_makemake` options and the `[General]/load-libs=` ini file option leave you enough room to do so.

8.2.4 Static vs shared OMNeT++ system libraries

Default linking uses the shared libraries. One reason you would want static linking is that debugging the OMNeT++ class library is more trouble with shared libraries. Another reason might be that you want to run the executable on another machine without having to worry about setting the `LD_LIBRARY_PATH` variable (which should contain the name of the directory where the OMNeT++ shared libraries are).

If you want static linking, find the

```
build_shared_libs=yes
```

line in the `configure.user` script and change it to

```
build_shared_libs=no
```

Then you have to re-run the configure script and rebuild everything:

```
./configure
make clean
make
```

8.3 Using Windows and Microsoft Visual C++

This is only a rough overview. Up-to-date, more detailed and more precise instructions can be found in the `doc/` directory of your OMNeT++ installation, in the file `Readme.MSVC`.

8.3.1 Installation

It is easiest to start with the binary, installer version. It contains all necessary software except MSVC, and you can get a working system up and running very fast.

Later you'll probably want to download and build the source distribution too. Reasons for that might be to compile the libraries with different flags, to debug into them, or to recompile with support for additional packages (e.g. Akaroa, MPI). Compilation should be painless (it takes a single `nmake -f Makefile.vc` command) after you get the different component directories right in `configuser.vc`. Additional software needed for the compilation is also described in `doc/`.

8.3.2 Building simulation models on the command line

OMNeT++ has an automatic MSVC makefile creator named `opp_nmakemake` which is probably the easier way to go. Its usage is very similar to the similarly named tool for Unix.

If you run `opp_nmakemake` in a directory of model sources, it collects all the names of all source files in the directory, and creates a makefile from them. The resulting makefile is called `Makefile.vc`.

To use `opp_nmakemake`, open a command window (*Start menu -> Run... –>* type `cmd`), then `cd` to the directory of your model and type:

```
opp_nmakemake
```

`opp_nmakemake` has several command-line options, mostly the same as the Unix version.

Then you can build the program by typing:

```
nmake -f Makefile.vc
```

The most common problem is that `nmake` (which is is part of MSVC) cannot be found because it is not in the path. You can fix this by running `vcvars32.bat`, which can be found in the MSVC `bin` directory (usually `C:\Program Files\Microsoft Visual Studio\VC98\Bin`).

8.3.3 Building simulation models from the MSVC IDE

You can also use the MSVC IDE for development. There is an MSVC wizard which will create project files for you, or you can start by copying one of the sample simulations. There is also an `AddNEDFileToProject` macro that can, well, add NED files to your project with the necessary custom build step (invoke nedc, etc.)

Some caveats (please read `doc/Readme.MSVC` for more!):

- **how to get the graphical environment**. By default, the sample simulations link with Cmdenv if you rebuild them from the IDE. To change to Tkenv, choose Build | Set active configuration from the menu, select "Debug-Tkenv" or "Release-Tkenv", then relink the executable.
- **can't find a usable init.tcl**. If you get this message, Tcl/Tk is missing the `TCL_LIBRARY` environment variable which is normally set by the installer. If you see this message, you need to set this variable yourself to the Tcl `lib/` directory.
- **changed compiler settings**. Changes since OMNeT++ 2.2: You'll need exception handling and RTTI turned ON, and stack size set to as low as 64K. See the readme file for rationale and more hints.
- **adding NED files**. After you added a `.ned` file to the project, you also have to add a `_n.cpp` file, and set a *Custom Build Step* for them:

```
Description: NED Compiling $(InputPath)
Command: nedc -s _n.cpp $(InputPath)
Outputs: $(InputName)_n.cpp
```

For msg files you need an analogous procedure.

- **file name extension**: as a gesture toward the free software community, MSVC refuses to treat `.cc` files as C++ sources, so first you have to rename them to `.cpp`. For the sample simulations this is done by `samples/cc2cpp.bat`.

Chapter 9

Running The Simulation

9.1 User interfaces

OMNeT++ simulations can be run under different user interfaces. Currenly, two 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 choose between them by linking one or the other into your simulation executable. (Creating the executable was described in chapter 8). Both user interfaces are supported on Unix and Windows platforms.

Common functionality in Tkenv and Cmdenv has been collected and placed into the Envir library, which can be thought of as the "common base class" for the two user interfaces.

The user interface is separated from the simulation kernel, and the two parts interact through a well-defined interface. This also means that, if needed, you can write your own user interface or embed an OMNeT++ simulation into your application without any change to models or the simulation library.

Configuration and input data for the simulation are described in a configuration file usually called `omnetpp.ini`. Some entries in this file apply to Tkenv or Cmdenv only, other settings are in effect regardless of the user interface. Both user interfaces accept command-line arguments, too.

The following sections explain `omnetpp.ini` and the common part of the user interfaces, describe Cmdenv and Tkenv in detail, then go on to specific problems.

9.2 The configuration file: omnetpp.ini

9.2.1 An example

For a start, let us see a simple `omnetpp.ini` file which can be used to run the Fifo1 sample simulation under Cmdenv.

```
[General]
network = fifonet1
sim-time-limit = 500000s
output-vector-file = fifo1.vec

[Cmdenv]
express-mode = yes

[Parameters]
# generate a large number of jobs of length 5..10 according to Poisson
fifonet1.gen.num_messages = 10000000
fifonet1.gen.ia_time = exponential(1)
fifonet1.gen.msg_length = intuniform(5,10)
# processing speeed of queue server
fifonet1.fifo.bits_per_sec = 10
```

The file is grouped into *sections* named `[General]`, `[Cmdenv]` and `[Parameters]`, each one containing several *entries*. The `[General]` section applies to both Tkenv and Cmdenv, and the entries in this case specify that the network named `fifonet1` should be simulated and run for 500,000 simulated seconds, and vector results should be written into the `fifo1.vec` file. The entry in the `[Cmdenv]` section tells Cmdenv to run the simulation at full speed and print periodic updates about the progress of the simulation. The `[Parameters]` section assigns values to parameters that did not get a value (or got `input` value) inside the NED files.

Lines that start with "#" or ";" are comments.

When you build the Fifo1 sample with Cmdenv and you run it by typing `fifo1` (or on Unix, `./fifo1`) on the command prompt, you should see something like this.

```
OMNeT++ Discrete Event Simulation  (C) 1992-2003 Andras Varga
See the license for distribution terms and warranty disclaimer
Setting up Cmdenv (command-line user interface)...

Preparing for Run #1...
Setting up network 'fifonet1'...
Running simulation...
** Event #0        T=0.0000000  ( 0.00s)   Elapsed: 0m  0s   ev/sec=0
** Event #100000   T=25321.99 ( 7h  2m)    Elapsed: 0m  1s   ev/sec=0
** Event #200000   T=50275.694 (13h 57m)   Elapsed: 0m  3s   ev/sec=60168.5
** Event #300000   T=75217.597 (20h 53m)   Elapsed: 0m  5s   ev/sec=59808.6
** Event #400000   T=100125.76 ( 1d  3h)   Elapsed: 0m  6s   ev/sec=59772.9
** Event #500000   T=125239.67 ( 1d 10h)   Elapsed: 0m  8s   ev/sec=60168.5
...
** Event #1700000  T=424529.21 ( 4d 21h)   Elapsed: 0m 28s   ev/sec=58754.4
** Event #1800000  T=449573.47 ( 5d  4h)   Elapsed: 0m 30s   ev/sec=59066.7
** Event #1900000  T=474429.06 ( 5d 11h)   Elapsed: 0m 32s   ev/sec=59453
** Event #2000000  T=499417.66 ( 5d 18h)   Elapsed: 0m 34s   ev/sec=58719.9
<!> Simulation time limit reached -- simulation stopped.

Calling finish() at end of Run #1...
*** Module: fifonet1.sink***
Total jobs processed: 9818
Avg queueing time:    1.8523
Max queueing time:    10.5473
```

```
Standard deviation:   1.3826

End run of OMNeT++
```

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 `fifo1.vec` contains vector data recorded during simulation (here, queueing times), and it can be processed using Plove or other tools.

9.2.2 The concept of simulation runs

OMNeT++ can execute several simulation runs automatically one after another. If multiple runs are selected, option settings and parameter values can be given either individually for each run, or together for all runs, depending in which section the option or parameter appears.

9.2.3 File syntax

The ini file is a text file consisting of entries grouped into different sections. The order of the sections doesn't matter. Also, if you have two sections with the same name (e.g. `[General]` occurs twice in the file), they will be merged.

Lines that start with "#" or ";" 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.

The size of the ini file (the number of sections and entries) is not limited. Currently there is a 1024-character limit on the line length, which *cannot* be increased by breaking up the line using backslashes. This limit might be lifted in future releases.

Example:

```
[General]
# this is a comment
foo="this is a single value \
for the foo parameter"

[General]  # duplicate sections are merged
bar="belongs to the same section as foo"
```

9.2.4 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
...
```

9.2.5 Sections

The following sections can exist:

Section	Description
`[General]`	Contains general settings that apply to all simulation runs and all user interfaces. For details, see section 9.2.6.
`[Run 1]`, `[Run 2]`, ...	Contains per-run settings. These sections may contain any entries that are accepted in other sections.
`[Cmdenv]`	Contains Cmdenv-specific settings. For details, see section 9.3.2
`[Tkenv]`	Contains Tkenv-specific settings. For details, see section 9.4.2
`[Parameters]`	Contains values for module parameters that did not get a value (or got `input` value) inside the NED files. For details, see section 9.5.1
`[OutVectors]`	Configures recording of output vectors. You can specify filtering by vector names and by simulation time (start/stop recording). For details, see section 9.5.2
`[DisplayStrings]`	Module display strings for Tkenv. For details, see section 9.5.3

9.2.6 The [General] section

The most important options of the `[General]` section are the following.

- The `ini-warnings` option can be used for "debugging" ini files: if enabled, it lists which options were searched for but not found.
- 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).
- The output file names can be set with the following options: `output-vector-file`, `output-scalar-file` and `snapshot-file`.

The full list of supported options follows. Almost every one these options can also be put into the `[Run` *n*`]` sections. Per-run settings have priority over globally set ones.

Entry and default value	Description
[General]	
`ini-warnings` = yes	Helps debugging of the ini file. If turned on, OMNeT++ prints out the name of the entries it that it wanted to read but they were not in the ini file.
`network` =	The name of the network to be simulated.
`snapshot-file` = omnetpp.sna	Name of the snapshot file. The result of each `snapshot()` call will be appended to this file.
`output-vector-file` = omnetpp.vec	Name of output vector file.
`output-scalar-file` = omnetpp.sca	Name of output scalar file.

`pause-in-sendmsg` = no	Only makes sense with step-by-step execution. If enabled, OMNeT++ will split `send()` calls to two steps.
`sim-time-limit` =	Duration of the simulation in simulation time.
`cpu-time-limit` =	Duration of the simulation in real time.
`random-seed` =	Random number seed for generator 0. Should be nonzero.
`gen0-seed` = `gen1-seed` = ...	Seeds for the given random number generator. They should be nonzero. `gen0-seed` is equivalent to `random-seed`.
`total-stack-kb` =	Specifies the total stack size (sum of all coroutine stacks) in kilobytes. You need to increase this value if you get the "Cannot allocate coroutine stack..." error.
`load-libs` =	Name of shared libraries (.so files) to load after startup. You can use it to load simple module code etc. Example: `load-libs = ../x25/x25.so ../lapb/lapb.so`
`netif-check-freq` =	Used with parallel execution.
`outputvectormanager-class` = `cFileOutputVectorManager`	Part of the Envir plugin mechanism: defines the name of the output vector manager class to be used to record data from output vectors. The class has to implement the `cOutputVectorManager` interface defined in `envirext.h`.
`outputscalarmanager-class` = `cFileOutputScalarManager`	Part of the Envir plugin mechanism: defines the name of the output scalar manager class to be used to record data passed to `recordScalar()`. The class has to implement the `cOutputScalarManager` interface defined in `envirext.h`.
`snapshotmanager-class` = `cFileSnapshotManager`	Part of the Envir plugin mechanism: defines the name of the class to handle streams to which `snapshot()` writes its output. The class has to implement the `cSnapshotManager` interface defined in `envirext.h`.

9.3 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 uses 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.

9.3.1 Command-line switches

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

`-h`	The program prints a short help message and the networks contained in the executable, then exits.
`-f` <*fileName*>	Specify the name of the configuration file. The default is `omnetpp.ini`. Multiple `-f` switches can be given; this allows you to partition your configuration file. For example, one file can contain your general settings, another one most of the module parameters, another one the module parameters you change often.
`-l` <*fileName*>	Load a shared object (`.so` file on Unix). Multiple `-l` switches are accepted. Your `.so` files may contain module code etc. By dynamically loading all simple module code and compiled network description (`_n.o` files on Unix) you can even eliminate the need to re-link the simulation program after each change in a source file. (Shared objects can be created with `gcc -shared...`)
`-r` <*runs*>	It specifies which runs should be executed (e.g. `-r 2,4,6-8`). This option overrides the `runs-to-execute=` option in the `[Cmdenv]` section of the ini file (see later).

All other options are read from the configuration file.

An example of running an OMNeT++ executable with the -h flag:

```
% ./fddi -h

OMNeT++ Discrete Event Simulation  (C) 1992-2003 Andras Varga
See the license for distribution terms and warranty disclaimer
Setting up Cmdenv (command-line user interface)...

Command line switches:
  -h            print this help and exit.
  -f <inifile>  use the given ini file instead of omnetpp.ini.
  -r <runs>     execute the specified runs in the ini file.
                <runs> is a comma-separated list of run numbers or
                run number ranges, for example 1,2,5-10.
  -l <library>  load the specified shared library on startup.
                The library can contain modules, networks, etc.

Available networks:
  FDDI1
  NRing
  TUBw
  TUBs

Available modules:
  FDDI_MAC
  FDDI_MAC4Ring
  ...

Available channels:


End run of OMNeT++
```

9.3.2 Cmdenv ini file options

Cmdenv can be executed in two modes, selected by the `express-mode` ini file entry:

- **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.

The full list of ini file options recognized by Cmdenv:

Entry and default value	Description
[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.
`express-mode`=yes/no (default: no)	Selects "normal" (debug/trace) or "express" mode.
`module-messages`=yes/no (default: yes)	In normal mode only: printing module ev« output on/off
`event-banners`=yes/no (default: yes)	In normal mode only: printing event banners on/off
`message-trace`=yes/no (default: no)	In normal mode only: print a line about each message sending (by `send()`,`scheduleAt()`, etc) and delivery on the standard output
`autoflush`=yes/no (default: no)	Call `fflush(stdout)` after each event banner or status update; affects both express and normal mode. Turning on autoflush can be useful with printf-style debugging for tracking down program crashes.
`status-frequency`=<integer> (default: 50000)	In express mode only: print status update every n events (on today's computers, and for a typical model, this will produce an update every few seconds, perhaps a few times per second)
`performance-display`=yes/no (default: yes)	In express mode only: 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.
`extra-stack` = 16384	Specifies the extra amount of stack (bytes) that is reserved for each `activity()` simple module when the simulation is linked with Cmdenv.

9.3.3 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 virtuall 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 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.

9.4 Tkenv: the graphical user interface

Features

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. The most important feaures are:

- message flow animation
- graphical display of statistics (histograms etc.) and output vectors during simulation execution
- separate window for each module's text output
- scheduled messages can be watched in a window as simulation progresses
- event-by-event, normal and fast execution
- labeled breakpoints
- inspector windows to examine and alter objects and variables in the model
- simulation can be restarted
- snapshots (detailed report about the model: objects, variables etc.)

Tkenv makes it possible to view simulation results (output vectors etc.) during execution. Results can be displayed as histograms and time-series diagrams. This can speed up the process of verifying the correct operation of the simulation program and provides a good environment for experimenting with the model during execution. When used together with `gdb` or `xxgdb`, Tkenv can speed up debugging a lot.

Tkenv is built with Tcl/Tk, and it work on all platforms where Tcl/Tk has been ported to: Unix/X, Windows, Macintosh. You can get more information about Tcl/Tk on the Web pages listed in the Reference.

9.4.1 Command-line switches

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

`-h`	The program prints a short help message and the networks contained in the executable, then exits.
`-f` <*fileName*>	Specify the name of the configuration file. The default is `omnetpp.ini`. Multiple `-f` switches can be given; this allows you to partition your configuration file. For example, one file can contain your general settings, another one most of the module parameters, another one the module parameters you change often.
`-l` <*fileName*>	Load a shared object (`.so` file on Unix). Multiple `-l` switches are accepted. Your `.so` files may contain module code etc. By dynamically loading all simple module code and compiled network description (`_n.o` files on Unix) you can even eliminate the need to re-link the simulation program after each change in a source file. (Shared objects can be created with `gcc -shared...`)

9.4.2 Tkenv ini file settings

Tkenv accepts several settings in the `[Tkenv]` section of the ini file. These settings can also be set within the graphical environment too, via menu items and dialogs.

Entry and default value	Description
[Tkenv]	
`default-run` = 1	Specifies which run Tkenv should set up automatically after startup. If there's no default-run= entry or the value is 0, Tkenv will ask which run to set up.
`use-mainwindow` = yes	Enables/disables writing *ev* output to the Tkenv main window.
`print-banners` = yes	Enables/disables printing banners for each event.
`breakpoints-enabled` = yes	Specifies whether the simulation should be stopped at each `breakpoint()` call in the simple modules.
`update-freq-fast` = 10	Number of events executed between two display updates when in *Fast* execution mode.
`update-freq-express` = 500	Number of events executed between two display updates when in *Express* execution mode.
`animation-delay` = 0.3s	Delay between steps when you slow-execute the simulation.
`animation-enabled` = yes	Enables/disables message flow animation.
`animation-msgnames` = yes	Enables/disables displaying message names during message flow animation.
`animation-msgcolors` = yes	Enables/disables using different colors for each message kind during message flow animation.
`animation-speed` = 1.0	Specifies the speed of message flow animation.
`extra-stack` = 32768	Specifies the extra amount of stack (bytes) that is reserved for each *`activity()`* simple module when the simulation is linked with Tkenv.

9.4.3 Using the graphical environment

Simulation running modes in Tkenv

Tkenv has the following modes for running the simulation :

- Step
- Run
- Fast run
- Express run

The running modes have their corresponding buttons on Tkenv's toolbar.

In **Step** mode, you can execute the simulation event-by-event.

In **Run** mode, the simulation runs with all tracing aids on. Message animation is active and inspector windows are updated after each event. Output messages are displayed in the main window and module output windows. You can stop the simulation with the Stop button on the toolbar. You can fully interact with the user interface while the simulation is running: you can open inspectors etc.

In **Fast** mode, animation is turned off. The inspectors and the message output windows are updated after each 10 events (the actual number can be set in Options | Simulation options and also in the ini file). Fast mode is several times faster than the Run mode; the speedup can get close to 10 (or the configured event count).

In **Express** mode, the simulation runs at about the same speed as with Cmdenv, all tracing disabled. Module output is not recorded in the output windows any more. You can interact with the simulation only once in a while (1000 events is the default as I recall), thus the run-time overhead of the user interface is minimal. You have to explicitly push the Update inspectors button if you want an update.

Tkenv has a status bar which is regularly updated while the simulation is running. The gauges displayed are similar to those in Cmdenv, described in section 9.3.3.

Inspectors

In Tkenv, objects can be viewed through inspectors. To start, choose Inspect | Network from the menu. Usage should be obvious; just use double-clicks and popup menus that are brought up by right-clicking. In Step, Run and Fast Run modes, inspectors are updated automatically as the simulation progresses. To make ordinary variables (int, double, char etc.) appear in Tkenv, use the `WATCH()` macro in the C++ code.

Tkenv inspectors also display the object pointer, and can also copy the pointer value to the clipboard. This can be invaluable for debugging: when the simulation is running under a debugger like gdb or the MSVC IDE, you can paste the object pointer into the debugger and have closer look at the data structures.

Configuring Tkenv

In case of nonstandard installation, it may be necessary to set the `OMNETPP_TKENV_DIR` environment variable so that Tkenv can find its parts written in Tcl script.

The default path from where the icons are loaded can be changed with the `OMNETPP_BITMAP_PATH` variable, which is a semicolon-separated list of directories and defaults to "*omnetpp-dir*/bitmaps;.;./bitmaps".

Embedding Tcl code into the executable

A significant part of Tkenv is written in Tcl, in several `.tcl` script files. The default location of the scripts is passed compile-time to `tkapp.cc`, and it can be overridden at run-time by the `OMNETPP_TKENV_DIR` environment variable. The existence of a separate script library can be inconvenient if you want to carry standalone simulation executables to different machines. To solve the problem, there is a possibility to compile the script parts into Tkenv.

The details: the `tcl2c` program (its C source is there in the Tkenv directory) is used to translate the `.tcl` files into C code (`tclcode.cc`), which gets included into `tkapp.cc`. This possibility is built into the makefiles and can be optionally enabled.

9.4.4 In Memoriam...

There used to be other windowing user interfaces which have been removed from the distribution:

- **TVEnv**. A Turbo Vision-based user interface, the first interactive UI for OMNeT++. Turbo Vision was an excellent character-graphical windowing environment, originally shipped with Borland C++ 3.1.
- **XEnv**. A GUI written in pure X/Motif. It was an experiment, written before I stumbled into Tcl/Tk and discovered its immense productivity in GUI building. XEnv never got too far because it was really very-very slow to program in Motif...

9.5 More about omnetpp.ini

9.5.1 Module parameters in the configuration file

Values for module parameters go into the `[Parameters]` or the `[Run 1]`, `[Run 2]` etc. sections of the ini file. The run-specific settings take precedence over the overall settings. Parameters that were assigned a (non-input) value in the NED file are not influenced by ini file settings.

Wildcards (*,?) can be used to supply values to several model parameters at a time. Filename-style (glob) and not regex-style pattern matching is used. Character ranges use curly braces instead of square brackets to avoid interference with the notation of module vectors: {a-zA-Z}. If a parameter name matches several wildcards-patterns, the first matching occurrence is used.

An example ini file:

```
# omnetpp.ini

[Parameters]
token.num_stations = 3
token.num_messages = 10000

[Run 1]
token.stations[*].wait_time = 10ms

[Run 2]
token.stations[0].wait_time = 5ms
token.stations[*].wait_time = 1000ms
```

9.5.2 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. Output vector configuration is given in the `[OutVectors]` section of the ini file, or in the `[Run 1]`, `[Run 2]` etc sections individually for each run. By default, all output vectors are turned on.

Entries configuring output vectors can be like that:

```
module-pathname.objectname.enabled = yes/no
module-pathname.objectname.interval = start..stop
module-pathname.objectname.interval = ..stop
module-pathname.objectname.interval = start..
```

The object name is the string passed to `cOutVector` in its constructor or with the `setName()` member function.

```
cOutVector eed("End-to-End Delay",1);
```

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:

```
#
# omnetpp.ini
#

[OutVectors]
*.interval = 1s..60s
*.End-to-End Delay.enabled = yes
*.Router2.*.enabled = yes
*.enabled = no
```

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.

9.5.3 Display strings

Display strings control the modules' graphical appearance in the Tkenv user interface. Display strings can be assigned to modules, submodules and gates (a connection's display string is stored in its "from" gate). Display strings can be hardcoded into the NED file or specified in the configuration file. (Hardcoded display strings take precedence over the ones given in ini files.) Format of display string are documented in section 4.8).

Display strings can appear in the `[DisplayStrings]` section of the ini file. They are expected as entries in one of the following forms:

```
moduletype = "..."
moduletype.submodulename = "..."

moduletype.inputgatename = "..."
moduletype.submodulename.outputgatename = "..."
```

As with parameter names, wildcards are allowed in module types, submodule and gate names.

9.5.4 Specifying seed values

As is was pointed out earlier, it is of great importance that different simulation runs and different random number sources within one simulation run use non-overlapping sequences of random numbers.

In OMNeT++, you have three choices:

1. Automatic seed selection.
2. Specify seeds in the ini file (with the help of the seedtool program, see later)
3. Manually set the seed from within the program.

If you decide for automatic seed selection, do not specify any seed value in the ini file. For the random number generators, OMNeT++ will automatically select seeds that are 1,000,000 values apart in the sequence. If you have several runs, each run is started with a fresh set of seeds that are 1,000,000 values apart from the seeds used for previous runs. Since the generation of new seed values is costly, OMNeT++ has a table of pre-calculated seeds (256 values); if they are all used up, OMNeT++ starts from the beginning of the table again.

Warning! Be aware that each time the simulation program is started, OMNeT++ starts assigning seeds from the beginning of the table. That is, if you execute Run 1, Run 2, Run 3 one at a time (e.g. from a shell script, using the Cmdenv -r command-line flag), all your runs will be executed using the *same* seeds! This behavior is almost surely not what you want, and it will be fixed in future versions of OMNeT++. Until then, it is a good idea to stick to manually generating seeds and explicitly adding them to the ini file.

Automatic seed selection may not be appropriate for you for several reasons. First, you may need more than 256 seeds values; or, if you use variance reduction techniques, you may want to use the same seeds for several simulation runs. In this case, there is a standalone program to generate appropriate seed values (`seedtool` will be discussed in Section 9.6), and you can specify the seeds explicitly in the ini file.

The following ini file explicitly initializes two of the random number generators, and uses different seed values for each run:

```
[Run 1]
gen0-seed = 1768507984
gen1-seed = 33648008

[Run 2]
gen0-seed = 1082809519
gen1-seed = 703931312
...
```

If you want the same seed values for all runs, you will write something like this:

```
[General]
gen0-seed = 1768507984
gen1-seed = 33648008
```

All other random number generators (2,3,...) will have their seeds automatically assigned. As a third way, you can also set the seed values from the code of a simple module using `genk_randseed()`, but I see no reason why you would want to do so.

9.6 Choosing good seed values: the seedtool utility

For selecting good seeds, the `seedtool` program can be used (it is in the utils directory). When started without command-line arguments, the program prints out the following help:

```
seedtool - part of OMNeT++, (c) 1992-2001 Andras Varga, TU Budapest
See the license for distribution terms and warranty disclaimer.

A tool to help select good random number generator seed values.
Usage:
  seedtool i seed         - index of 'seed' in cycle
  seedtool s index        - seed at index 'index' in cycle
  seedtool d seed1 seed2  - distance of 'seed1' and 'seed2' in cycle
  seedtool g seed0 dist   - generate seed 'dist' away from 'seed0'
  seedtool g seed0 dist n - generate 'n' seeds 'dist' apart, starting
                            at 'seed0'
  seedtool t              - generate hashtable
  seedtool p              - print out hashtable
```

The last two options, p and t were used internally to generate a hash table of pre-computed seeds that greatly speeds up the tool. For practical use, the g option is the most important. Suppose you have 4 simulation runs that need two independent random number generators each and you want to start their seeds at least 10,000,000 values apart. The first seed value can be simply 1. You would type the following command:

```
C:\OMNETPP\UTILS> seedtool g 1 10000000 8
```

The program outputs 8 numbers that can be used as random number seeds:

```
1768507984
33648008
1082809519
703931312
1856610745
784675296
426676692
1100642647
```

You would specify these seed values in the ini file.

9.7 Repeating or iterating simulation runs

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.

The next sections are only for Unix users. We'll use the Unix 'Bourne' shell (`sh`, `bash`) to write the control script. If you'd prefer Matlab/Octave, the `contrib/octave/` directory contains example scripts (contributed by Richard Lyon).

9.7.1 Executing several runs

In simple cases, you may define all simulation runs needed in the `[Run 1]`, `[Run 2]`, etc. sections of `omnetpp.ini`, and invoke your simulation with the -r flag each time. The -f flag lets you use a file name different from `omnetpp.ini`.

The following script executes a simulation named `wireless` several times, with parameters for the different runs given in the `runs.ini` file.

```
#! /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
```

9.7.2 Variations over parameter values

It may not be practical to hand-write descriptions of all runs in an ini file, especially if there are many parameter settings to try, or you want to try all possible combinations of two or more parameters. The solution might be to generate only a small fraction of the ini file with the variable parameters, and use it via ini file inclusion. For example, you might write your `omnetpp.ini` like this:

```
[General]
network = Wireless

[Parameters]
Wireless.n = 10
...   # other fixed parameters
include params.ini  # include variable part
```

And have the following as control script. It uses two nested loops to explore all possible combinations of the *alpha* and *beta* parameters. Note that `params.ini` is created by redirecting the `echo` output into file, using the > and » operators.

```
#! /bin/sh
for alpha in 1 2 5 10 20 50; do
   for beta in 0.1 0.2 0.3 0.4 0.5; do
       echo "Wireless.alpha=$alpha" > params.ini
       echo "Wireless.beta=$beta" >> params.ini
       ./wireless
   done
done
```

As a heavy-weight example, here's the "runall" script of Joel Sherrill's *File System Simulator*. It also demonstrates that loops can iterate over string values too, not just numbers. (`omnetpp.ini` includes the generated `algorithms.ini`.)

Note that instead of redirecting every `echo` command to file, they are grouped using parentheses, and redirected together. The net effect is the same, but you can spare some typing this way.

```
#! /bin/bash
#
# This script runs multiple variations of the file system simulator.
#
all_cache_managers="NoCache FIFOCache LRUCache PriorityLRUCache..."
all_schedulers="FIFOScheduler SSTFScheduler CScanScheduler..."

for c in ${all_cache_managers}; do
  for s in ${all_schedulers}; do
  (
    echo "[Parameters]"
    echo "filesystem.generator_type = \"GenerateFromFile\""
    echo "filesystem.iolibrary_type = \"PassThroughIOLibrary\""
    echo "filesystem.syscalliface_type = \"PassThroughSysCallIface\""
    echo "filesystem.filesystem_type = \"PassThroughFileSystem\""
    echo "filesystem.cache_type = \"${c}\""
    echo "filesystem.blocktranslator_type = \"NoTranslation\""
```

```
    echo "filesystem.diskscheduler_type = \"${s}\""
    echo "filesystem.accessmanager_type = \"MutexAccessManager\""
    echo "filesystem.physicaldisk_type = \"HP97560Disk\""
  ) >algorithms.ini

  ./filesystem
  done
done
```

9.7.3 Variations over seed value (multiple independent runs)

The same kind of control script can be used if you want to execute several runs with different random seeds. The following code does 500 runs with independent seeds. (`omnetpp.ini` should include `parameters.ini`.)

The seeds are 10 million numbers apart in the sequence (`seedtool` parameter), so one run should not use more random numbers than this, otherwise there will be overlaps in the sequences and the runs will not be independent.

```
#! /bin/sh
seedtool g 1 10000000 500 > seeds.txt
for seed in ‘cat seeds.txt‘; do
   (
     echo "[General]"
     echo "random-seed = ${seed}"
     echo "output-vector-file = xcube-${seed}.vec"
   ) > parameters.ini
   ./xcube
done
```

9.8 Akaroa support: Multiple Replications in Parallel

9.8.1 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 [PJL02] and be horrified.

A possible solution is to look at the statistics while the simulation is running, and decide at runtime when enough data have been collected for the results to have reached the required

accuracy. One possible criterion is given by the confidence level, more precisely, by its width relative to the mean. But ex ante it is unknown how many observations have to be collected to achieve this level – it must be determined runtime.

9.8.2 What is Akaroa

Akaroa [EPM99] addresses the above problem. According to its authors, Akaroa (Akaroa2) is a "fully automated simulation tool designed for running distributed stochastic simulations in MRIP scenario" in a cluster computing environment.

MRIP stands for *Multiple Replications in Parallel*. In MRIP, the computers of the cluster run independent replications of the whole simulation process (i.e. with the same parameters but different seed for the RNGs (random number generators)), generating statistically equivalent streams of simulation output data. These data streams are fed to a global data analyser responsible for analysis of the final results and for stopping the simulation when the results reach a satisfactory accuracy.

The independent simulation processes run independently of one another and continuously send their observations to the central analyser and control process. This process *combines* the independent data streams, and calculates from these observations an overall estimate of the mean value of each parameter. Akaroa2 decides by a given confidence level and precision whether it has enough observations or not. When it judges that is has enough observations it halts the simulation.

If n processors are used, the needed simulation execution time is usually n times smaller compared to a one-processor simulation (the required number of observations are produced sooner). Thus, the simulation would be sped up approximately in proportion to the number of processors used and sometimes even more.

Akaroa was designed at the University of Canterbury in Christchurch, New Zealand and can be used free of charge for teaching and non-profit research activities.

9.8.3 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 of course not detailed enough help you set up and successfully use Akaroa – for that you need to read the Akaroa manual. The purpose was rather to give you the "flavour" of using it.

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 (except for RNGs, see later) – it continues to write the observations into output vectors (`cOutVector` objects, see chapter 7). You can place some of the output vectors under Akaroa control.

You need to add the following to `omnetpp.ini`:

```
[General]
outputvectormanager-class="AkOutputVectorManager"
```

The above line replaces the normal output vector handler by its Akaroa-enabled version, using the Envir plugin interface.

Also, you have to specify which output vectors you want to be under Akaroa control. By default, all output vectors are under Akaroa control; the

```
<modulename>.<vectorname>.akaroa=false
```

setting can be used to make Akaroa ignore specific vectors. If you only want a few vectors be placed under Akaroa, you can use the following "trick":

```
<modulename>.<vectorname1>.akaroa=true
<modulename>.<vectorname2>.akaroa=true
...
*.*.akaroa=false
```

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. Your can prevent this from happening using the `fname-append-host` ini file entry:

```
[General]
fname-append-host=yes
```

When turned on, it appends the host name to the names of the output files (output vector, output scalar, snapshot files).

Random number generation

Another important point is that you cannot use the random number generators provided by OMNeT++, but rather you have to obtain random numbers from Akaroa.

Unfortunately, OMNeT++'s RNGs do not (yet) know anything about Akaroa. Remember, all simulation processes are run with exactly the same configuration – including the same

RNG seeds. That is, if you'd use OMNeT++'s own RNGs, all simulation processes would use *identical* random number streams, thus executing *exactly the same sequence of events!* This is clearly not what you want or what Akaroa expects.

Akaroa provides the following random number functions. The underlying 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.

Future versions of OMNeT++ will wrap the random number functions of Akaroa, thus requiring no modifications to simulation programs.

```
#include <akaroa/distribution.H>

real Uniform(real a, real b);
long UniformInt(long n0, long n1);
long Binomial(long n, real p);
real Exponential(real m);
real Erlang(real m, real s);
real HyperExponential(real m, real s);
real Normal(real m, real s);
real LogNormal(real m, real s);
long Geometric(real m);
real HyperGeometric(real m, real s);
long Poisson(real m);
real Weibull(real alpha, real beta);
```

9.9 Typical problems

9.9.1 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. You should increase the stack for that module type. You can call the `stackUsage()` 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 extra-stack-for-envir, the sum of the two typically around 32K. 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.

9.9.2 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* (enventually 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 9.3.3. 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's still possible you're leaking other `cObject`-based objects. You can also find them using Tkenv's *Inspect|From list of all objects...* function.

If you're leaking non-`cObject`-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. [1]

[1] Alternatively, you can go through the full code, review it looking for bugs. In my experience, the latter one has proved to be far more efficient than using any kind of memory debugger.

The number one of these tools is *Purify* from Rational Software. This is a commercial tool. Not particularly cheap, but it has very good reputation and proved its usefulness many times.

There are several open source tools you can try. The best seems to be *Valgrind* used by the KDE people. Other good ones are *NJAMD*, *MemProf*, *MPatrol* and *dmalloc*, while *ElectricFence* seems to be included in most Linux distributions. 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.

Poor man's memory leak debugger

However, if you don't have such tools, you can use the basic heap debugging code in Cmdenv. It is disabled by default; to turn it on, you have to uncomment the `#defines` in `src/envir/cmdenv/heap.cc`:

`HEAPCHECK`	checks heap on new/delete
`COUNTBLOCKS`	counts blocks on heap and tells it if none left
`ALLOCTABLE`	remembers pointers and reports heap contents if only LASTN blocks remained
`DISPLAYALL`	reports every new/delete
`DISPSTRAYS`	reports deleting of pointers that were not registered by operator new or that were deleted since then
`BKPT`	calls a function at a specified new/delete; you can set a breakpoint to that function

If `COUNTBLOCKS` is turned on, you should see the `[heap.cc-DEBUG:ALL BLOCKS FREED OK]` message at the end of the simulation. If you do not see it, it means that some blocks have not been freed up properly, that is, your simulation program is likely to have memory leaks.

Chapter 10

Analyzing Simulation Results

10.1 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 dopped packets, etc. – anything that's 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 7.9.1). 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 9.5.2).

All `cOutVector` objects write to the same, common file. The following sections describe the format of the file, and how you can process them.

10.2 Plotting output vectors with Plove

10.2.1 Plove features

Typically, you'll get output vector files as a result of a simulation. Data written to `cOutVector` objects from simple modules go to output vector files. Normally, you use Plove to look into output vector files and plot vectors in it.

Plove is a handy tool for plotting OMNeT++ output vectors. It uses Gnuplot to do the actual work. You can specify the drawing style (lines, dots etc) for each vector as well as set the most frequent drawing options like axis bounds, scaling, titles and labels etc. You can save the gnuplot graphs to files (postscript, latex, pbm etc) with a click. Plove can also generate standalone shell scripts that plot output vectors in much the same way Plove does itself. These scripts can be used for batch processing or to debug filters (see later). Plove does not take away any of gnuplot's flexibility – you can embed your own gnuplot commands to customize the output.

Filtering the results before plotting is possible. Filters can do averaging, truncation of extreme values, smoothing, they can do density estimation by calculating histograms etc. Some filters are built in, and you can easily create new filters or modify the existing ones. Filters can be incorporated in one of three ways: as `awk` expressions, as `awk` programs and as external filter programs. Filters can be parameterized. Multiple filters for the same vector is not currently supported; also, you cannot currently feed several vectors into a single filter.

Plove does not create temporary files, so you don't need to worry about disk space: if the

output vector is there, Plove can plot it for you. Moreover, it can also work with gzipped vector files without extracting them – just make sure you have zcat.

Plove never modifies the output vector files themselves.

On startup, Plove automatically reads the `.ploverc` file in your home directory. The file contains general gnuplot settings, the filter configuration etc. (that is, the stuff from the Options menu).

10.2.2 Usage

First, you load an output vector file (`.vec`) into the left pane. You can also load gzipped vector files (`.vec.gz`) without having to decompress them. You can copy vectors from the left pane to the right pane by clicking the right arrow icon in the middle. The large PLOT button will plot the *selected* vectors in the right pane. Selection works as in Windows: dragging and shift+left click selects a range, and ctrl+left click selects/deselects individual items. To adjust drawing style, change vector title or add filter, push the Options... button. This works for several selected vectors too. Plove accepts nc/mc-like keystrokes: F3, F4, F5, F6, F8, grey '+' and grey '*'.

The left pane works as a general storage for vectors you're working with. You can load several vector files, delete vectors you don't want to deal with, rename them etc. All this will not affect the vector files on disk. In the right pane, you can duplicate vectors if you want to filter the vector and also keep the original. If you set the right options for a vector but temporarily do not want it to hang around in the right pane, you can put it back into the left pane for storage.

10.2.3 Writing filters

Filters get an output vector on their standard input (as plain text, with the timestamp being the second and the value being the third field on each line), do some processing to it and write the result to the standard output.

Filters can be incorporated in one of three ways: as `awk` expressions, as `awk` programs or as external programs. An 'awk expression' filter means assembling and launching a command like this:

```
cat foobar.vec | awk '{$3 = <expression>; print}' | ...
```

An `awk` program filter means running the following command:

```
cat foobar.vec | awk '{<program>}' | ...
```

The third type of filters is used like this:

```
cat foobar.vec | <program> <parameters> | ...
```

Before the filter pipeline is launched, the following substitutions are performed on the `awk` scripts:

- **t** gets substituted to $2, which is the simulation time (the second column in the output vector file)
- **x** gets substituted to $3, the actual value (the third column)

The parameters of the form `$(paramname)` are also replaced with their actual value.

For example, if you want to add 1 to all values, you can use the `awk` expression filter `x+1`. It will turn into the following awk script:

```
awk '{$3 = $3+1}; print'
```

When you want to shift the vector by a used-defined DT time, you can create the following `awk` program filter:

```
{t += $(DT); print}
```

To plot the mean on *(0,t)*, you'd write

```
{sum+=x; x=sum/++n; print}
```

Do not forget the print statement, or your filter will not output anything and the gnuplot graph will be empty.

Filters are automatically saved into and loaded from the `/.ploverc` file.

10.3 Format of output vector files

An output vector file contains several series of data produced during simulation. The file is textual, it looks like this:

```
mysim.vec:
vector 1   "subnet[4].term[12]"  "response time"  1
1  12.895  2355.66666666
1  14.126  4577.66664666
vector 2   "subnet[4].srvr"  "queue length"  1
2  16.960  2.00000000000.63663666
1  23.086  2355.66666666
2  24.026  8.00000000000.44766536
```

There are label lines (beginning with vector) and data lines.

A vector line introduces a new vector. Its columns are: vector ID, module of creation, name of `cOutVector` object, multiplicity of data (single numbers or pairs will be written).

Lines beginning with numbers are data lines. The columns: vector ID, current simulation time, and one or two double values.

10.4 Working without Plove

10.4.1 Extracting vectors from the file

You can use the Unix `grep` tool to extract a particular vector from the file. As the first step, you must find out the ID of the vector. You can find the appropriate vector line with a text editor or you can use `grep` for this purpose:

```
% grep "queue length" vector.vec
```

Or, you can get the list of all vectors in the file by typing:

```
% grep ^vector vector.vec
```

This will output the appropriate vector line:

```
vector 6  "subnet[4].srvr"  "queue length"  1
```

Pick the vector ID, which is 6 in this case, and grep the file for the vector's data lines:

```
grep ^6 vector.vec > vector6.vec
```

Now, `vector6.vec` contains the appropriate vector. The only potential problem is that the vector ID is there at the beginning of each line and this may be hard to explain to some programs that you use for post-processing and/or visualization. This problem is eliminated by the OMNeT++ `splitvec` utility (written in `awk`), to be discussed in the next section.

10.4.2 Using splitvec

The `splitvec` script (part of OMNeT++) breaks the vector file into several files which contain one vector each:

```
% splitvec mysim.vec
```

creates several files: mysim1.vec, mysim2.vec etc.

mysim1.vec:

```
# vector 1  "subnet[4].term[12]"  "response time"  1
12.895  2355.66666666
14.126  4577.66664666
23.086  2355.66666666
```

mysim2.vec:

```
# vector 2  "subnet[4].srvr"  "queue length"  1
16.960  2.00000000000.63663666
24.026  8.00000000000.44766536
```

As you can see, the vector ID is gone.

The files can be further processed with math packages, or read by analysis or spreadsheet programs which provide numerous ways to display data as diagrams, do calculations on them etc. One could use for example Gnuplot, Matlab, Excel, etc.

10.4.3 Visualization under Unix

Two programs are in common use: Gnuplot and Xmgr. Both are free and both have their good and bad sides; we'll briefly discuss them. There are innumerable tutorials and documentation about them on the Web; some of them you will find among the References.

Both programs can eat files produced by `splitvec`. Both programs can produce output in various forms: on screen, in Postscript format, printer files, Latex output etc. For DTP purposes, Postscript seems to be the most appropriate. On Windows, the easiest way is to copy the picture to the clipboard from the Gnuplot window's system menu.

Gnuplot has an interactive command interface. To get the vectors in `mysim1.vec` and `mysim4.vec` plotted in the same graph, you can type:

```
plot "mysim1.vec" with lines, "mysim4.vec" with lines
```

To adjust the y range, you would type:

```
set yrange [0:1.2]
replot
```

There are several commands to adjust ranges, plotting style, labels, scaling etc. Gnuplot can also plot 3D graphs. Gnuplot is also available for Windows and other platforms. Gnuplot also has a simple graphical interactive user interface called PlotMTV. However, we recommend that you use OMNeT++'s Plove tool, described in an earlier section.

Xmgr is an X/Motif based program, with a menu-driven graphical interface. You load the appropriate file by selecting in a dialog box. The icon bar and menu commands can be used to customize the graph. Some say that Xmgr can produce nicer output that Gnuplot and it is easier to use. Xmgr cannot do 3D and only runs on Unixes with X and Motif installed. Xmgr also has a batch interface so you can use it from scripts too.

Chapter 11

Parallel Execution

11.1 OMNeT++ support for parallel execution

11.1.1 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 to several LPs that will be simulated independently on different hosts or processors. Each LP will have its own local Future Event Set, thus they will maintain local simulation times. The main issue with parallel simulations is keeping LPs synchronized in order to avoid violating 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 solve the causality problem outlined above:

1. **Conservative algorithms** prevents incausalities from happening. The Null Message Algorithm exploits knowledge about when LPs send messages to other LPs, and uses 'null' messages to propagate this info to other LPs. If a LP knows it won't receive any messages from other LPs until $t + \Delta t$ simulation time, it may advance until $t + \Delta t$ without the need for external synchronization. Conservative simulation tends to converge to sequential simulation (slowed down by communication between LPs) if there's not enough parallelism in the model, or parallelism is not exploited by sending enough 'null' messages.

2. **Optimistic synchronization** allows incausalities to occur, but detects and repairs them. Repairing involves rollbacks to a previous state, sending out anti-messages to cancel messages sent out during the period that is being rolled back, etc. Optimistic synchronization is extremely difficult to implement, because it requires periodic state saving and the ability to restore previous states. In any case, implementing optimistic synchronization in 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.

11.1.2 In work

Parallel simulation support in OMNeT++ has recently been reimplemented. The new code is currently being refined and tested, and it will be available in the following releases.

Chapter 12

Customization and Embedding

12.1 Architecture

OMNeT++ has a modular architecture. The following diagram shows the high-level architecture of OMNeT++ simulations:

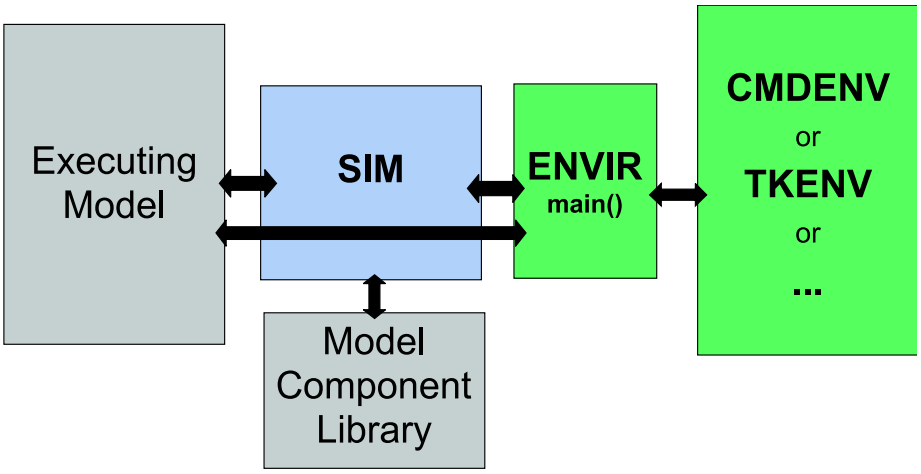

Figure 12.1: Architecture of OMNeT++ simulation programs

The rectangles in the picture represent functional blocks:

- **Sim** is the simulation kernel and class library. Sim exists as a library you link your simulation program with. [1]
- **Envir** is another library which contains all code that is common for 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 implementing a new user interface in addition to Cmdenv and Tkev, or by replacing Envir with another implementation of `ev` (see sections 12.5.3 and 12.2.)

[1] Use of dynamic (shared) libraries is also possible, but for simplicity we'll use the word *linking* here.

- **Cmdenv and Tkenv** are specific user interface implementations. A concrete simulation is linked with either Cmdenv or Tkenv.

- The **Model Component Library** is the simple module definitions and their C++ implementations, compound module types, channels, networks, message types and in general everything that belong 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 functional blocks interact with each other:

- **Executing Model vs 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 vs 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 vs Envir**. The `ev` object, logically part of Envir, is the facade of the user interface towards the executing model. The model uses `ev` for writing debug logs (`ev«`, `ev.printf()`).

- **Sim vs 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 vs Tkenv and Cmdenv**. Envir defines `TOmnetApp` as a base class for user interfaces, and Tkenv and Cmdenv both subclass from `TOmnetApp`. The `main()` function provided as part of Envir determines the appropriate user interface class (subclassed from `TOmnetApp`), creates an instance and runs it – whatever happens next (opening a GUI window or running as a command-line program) is decided in the `run()` method of the appropriate `TOmnetApp` subclass. Sim's or the model's calls on the `ev` object are simply forwarded to the `TOmnetApp` instance. Envir presents a framework and base functionality to Tkenv and Cmdenv via the methods of `TOmnetApp` and some other classes.)

12.2 Embedding OMNeT++

Embedding is a special issue. You probably do not want to keep the appearance of the simulation program, so you do not want Cmdenv and Tkenv. You may or may not want to keep Envir.

What you'll absolutely need for a simulation to run is the Sim library. You can keep Envir if its philosophy and the infrastructure it provides (`omnetpp.ini`, certain command-line options etc.) fit into your design. Then your application, the embedding program will take the place of Cmdenv and Tkenv.

If Envir does not fit your needs (for example, you want the model parameters to come from a database not from `omnetpp.ini`), then you have to replace it. Your Envir replacement (the embedding application, practically) must implement the `cEnvir` member functions from `envir/cenvir.h`, but you have full control over the simulation.

Normally, code that sets up a network or builds the internals of a compound module comes from compiled NED source. You may not like the restriction that your simulation program can only simulate networks whose setup code is linked in. No problem; your program can contain pieces of code like what is generated by nedc and then it can build any network whose components (primarily the simple modules) are linked in. Moreover, it is possible to write an integrated environment where you can put together a network using a graphical editor and right after that you can run it, without intervening NED compilation and linkage.

12.3 Sim: the simulation kernel and class library

There is little to say about Sim here, since chapters 5 and 7, and part of chapter 6 are all on this topic. Classes covered in those chapters are documented in more detail in the API Reference generated by Doxygen. What we can do here is documenting some internals that were not appropriate to be treated in the general chapters.

The source code for the simulation kernel and class library reside in the `src/sim/` subdirectory.

12.3.1 The global simulation object

The global `simulation` object is an instance of `cSimulation`. It stores the model, and encapsulated much of the functionality of setting up and running a simulation model.

`simulation` has two basic roles:

- stores modules of the executing model
- holds the future event set (FES) object

12.3.2 The coroutine package

The coroutine package is in fact two coroutine packages:

- Portable coroutine package creates all coroutine stacks inside the main stack. It is based on Kofoed's solution[Kof95]. It allocates stack by deep-deep recursions and then plays with `setjmp()` and `longjmp()` to switch from one another.
- On Windows, the Fiber functions (`CreateFiber()`, `SwitchToFiber()`, etc) are used, which are part of the standard Win32 API.

The coroutines are represented by the `cCoroutine` class. `cSimpleModule` has `cCoroutine` as one a base class.

12.4 The Model Component Library

All model components (simple module definitions and their C++ implementations, compound module types, channels, networks, message types, etc.) that you compile and link into a simulation program are registered in the Model Component Library. Any model that has all its necessary components in the component library of simulation program can be run by that simulation program.

If your simulation program is linked with Cmdenv or Tkenv, you can have the contents of its component library printed, using the -h switch.

```
% ./fddi -h

OMNeT++ Discrete Event Simulation  (C) 1992-2003 Andras Varga
...
Available networks:
  FDDI1
  NRing
  TUBw
  TUBs

Available modules:
  FDDI_MAC
  FDDI_MAC4Ring
  ...

Available channels:
  ...
End run of OMNeT++
```

Information on components are kept on registration lists. There are macros for registering components (that is, for adding them to the registeration lists): `Define_Module()`, `Define_Module_Like()`, `Define_Network()`, `Define_Function()`, `Register_Class()`, and a few others. For components defined in NED files, the macro calls are generated by the NED compiler; in other cases you have to write them in your C++ source.

The machinery for managing the registrations lists are part of the Sim library. Registration lists are implemented as global objects.

The registration lists are:

List object	Macro that creates a mer Class of members	Function

`cHead networks;`	`Define_Network()`
	
	`cNetworkType`
		List of available networks. A `cNetworkType` object holds a pointer to a function that can build up the network. `Define_Network()` macros occur in the code generated by the NEDC compiler.

cHead modtypes;	Define_Module(), Define_Module_Like(), cModuleType	List of available module types. A cModuleType object knows how to create a module of a specific type. If it is compound, it holds a pointer to a function that can build up the inside. Usually, Define_Module() macros for compound modules occur in the code generated by the NEDC compiler; for simple modules, the Define_Module() lines are added by the user.
cHead classes;	Register_Class() cClassRegister	List of available classes of which the user can create an instance. A cClassRegister object knows how to create an (empty) object of a specific class. The list is used by the createOne() function that can create an object of any (registered) type from a string containing the class name. (E.g. ptr = createOne("cArray") creates an empty cArray.) Register_Class() macros are present in the simulation source files for existing classes; has to be written by the user for new classes.
cHead functions;	Define_Function() cFunctionType	List of functions taking doubles and returning a double (see type MathFuncNoArg...MathFunc3Args). A cFunctionType object holds a pointer to the function and knows how many arguments it takes.
cHead linktypes;	Define_Link() cLinkType	List of link types. A cLinkType object knows how to create cPar objects representing the delay, error and datarate attributes for a channel. Define_Link() macros occur in the code generated by the NEDC compiler, one for each channel definition.

12.5 Envir, Tkenv and Cmdenv

The source code for the user interface of OMNeT++ resides in the src/envir/ directory (common part) and in the src/cmdenv/, src/tkenv/ directories.

The classes in the user interface are *not* derived from cObject, they are completely separated from the simulation kernel.

12.5.1 The main() function

The main() function of OMNeT++ simply sets up the user interface and runs it. Actual simulation is done in cEnvir::run() (see later).

12.5.2 The cEnvir interface

The cEnvir class has only one instance, a global object called ev:

```
cEnvir ev;
```

`cEnvir` basically a facade, its member functions contain little code. `cEnvir` maintains a pointer to a dynamically allocated simulation application object (derived from `TOmnetApp`, see later) which does all actual work.

`cEnvir` member functions deal with four basic tasks:

- I/O for module activities; actual implementation is different for each user interface (e.g. stdin/stdout for Cmdenv, windowing in Tkenv)
- setting up and running the simulation application
- provides functions called by simulation kernel objects to get information (for example, get module parameter settings from the configuration file)
- provides functions called by simulation kernel objects to notify the user interface of some events. This is especially important for windowing user interfaces (Tkenv), because the events are like this: an object was deleted so its inspector window should be closed; a message was sent so it can be displayed; a breakpoint was hit.

12.5.3 Customizing Envir

Certain aspects of Envir can be customized via plugin interfaces. Plugin interfaces are presented in the form of C++ abstract classes that you have to implement, register via the `Register_Class()` macro, and finally tell Envir to use them via `omnetpp.ini` entries.

The following plugin interfaces are supported:

- `cOutputScalarManager`. It handles recording the scalar output data, output via the cModule::recordScalar() family of functions. The default output scalar manager is `cFileOutputScalarManager`, defined in the Envir library.
- `cOutputVectorManager`. It handles recording the output for `cOutVector` objects. The default output vector manager is `cFileOutputVectorManager`, defined in the Envir library.
- `cSnapshotManager`. It provides an output stream to which snapshots are written (see section 7.10.3). The default snapshot manager is `cFileSnapshotManager`, defined in the Envir library.

The above interfaces are documented in the API Reference, generated by Doxygen.

The corresponding ini file entries that allow you to select your plugin classes are `outputvectormanager-class`, `outputscalarmanager-class` and `snapshotmanager-class`, documented in section 9.2.6.

12.5.4 Implementation of the user interface: simulation applications

The base class for simulation application is `TOmnetApp`. Specific user interfaces such as `TCmdenv`, `TOmnetTkApp` are derived from `TOmnetApp`.

`TOmnetApp`'s member functions are almost all virtual.

- Some of them implement the `cEnvir` functions (described in the previous section)
- Others implement the common part of all user interfaces (for example: reading options from the configuration files; making the options effective within the simulation kernel)

- The `run()` function is pure virtual (it is different for each user interface).

`TOmnetApp`'s data members:

- a pointer to the object holding configuration file contents (type `cInifile`);
- the options and switches that can be set from the configuration file (these members begin with `opt_`)

Concrete simulation applications:

- add new configuration options
- provide a `run()` function
- implement functions left empty in `TOmnetApp` (like `breakpointHit()`, `objectDeleted()`).

Appendix A

NED Language Grammar

The NED language, the network topology description language of OMNeT++ will be given using the extended BNF notation.

Space, horizontal tab and new line characters counts 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 only distinguishes between lower and upper case letters in names, but not in keywords.

In this description, the {xxx...} notation stands for one or more xxx's separated with spaces, tabs or new line characters, and {xxx,,,} stands for one or more xxx's, separated with a comma and (optionally) spaces, tabs or new line characters.

For ease of reading, in some cases we use textual definitions. The *networkdescription* symbol is the sentence symbol of the grammar.

notation	meaning
[a]	0 or 1 time a
{a}	a
{a,,,}	1 or more times a, separated by commas
{a...}	1 or more times a, separated by spaces
a|b	a or b
'a'	the character a
bold	keyword
italic	identifier

```
networkdescription ::=
    { definition... }

definition    ::=
      include
    | channeldefinition
    | simpledefinition
    | moduledefinition
    | networkdefinition

include ::=
    INCLUDE { fileName ,,, } ;

channeldefinition ::=
```

```
    CHANNEL channeltype
     [ DELAY numericvalue ]
     [ ERROR numericvalue ]
     [ DATARATE numericvalue ] $^******$
    ENDCHANNEL

simpledefinition ::=
    SIMPLE simplemoduletype
     [ machineblock ]
     [ paramblock ]
     [ gateblock ]
    ENDSIMPLE [ simplemoduletype ]

moduledefinition ::=
    MODULE compoundmoduletype
     [ machineblock$^*$ ]
     [ paramblock ]
     [ gateblock ]
     [ submodblock ]
     [ connblock ]
    ENDMODULE [ compoundmoduletype ]

moduletype ::=
    simplemoduletype | compoundmoduletype

machineblock ::=
    MACHINES: { machine ,,, } ;

paramblock ::=
    PARAMETERS: { parameter ,,, } ;

parameter ::=
    parametername
    | parametername : CONST [ NUMERIC ]
    | parametername : STRING
    | parametername : BOOL
    | parametername : CHAR
    | parametername : ANYTYPE

gateblock ::=
    GATES:
     [ IN: { gate ,,, } ; ]
     [ OUT: { gate ,,, } ; ]
gate ::=
    gatename [ '[]' ]

submodblock ::=
    SUBMODULES: { submodule... }

submodule ::=
    { submodulename : moduletype [ vector ]
     [ on_block$^*$... ]
     [ substparamblock... ]
     [ gatesizeblock... ] }
```

```
  | { submodulename : parametername [ vector ] LIKE moduletype
      [ on_block$^*$... ]
      [ substparamblock... ]
      [ gatesizeblock... ] }

on_block$^*$ ::=
    ON [ IF expression ]: { on_machine ,,, } ;

substparamblock    ::=
    PARAMETERS [ IF expression ]:
      { substparamname = substparamvalue,,, } ;

substparamvalue ::=
    ( [ ANCESTOR ] [ REF ] name )
    | parexpression

gatesizeblock ::=
    GATESIZES [ IF expression ]:
      { gatename vector ,,, } ;

connblock ::=
    CONNECTIONS [ NOCHECK ]: { connection ,,, } ;

connection ::=
     normalconnection | loopconnection

loopconnection ::=
    FOR { index... } DO
      { normalconnection ,,, } ;
    ENDFOR

index ::=
    indexvariable '=' expression ''...'' expression

normalconnection ::=
     { gate { --> | <-- } gate [ IF expression ]}
   | {gate --> channel --> gate [ IF expression ]}
   | {gate <-- channel <-- gate [ IF expression ]}

channel ::=
     channeltype
    | [ DELAY expression ] [ ERROR expression ] [ DATARATE expression ]
        $^******$

gate ::=
    [ modulename [vector]. ] gatename [vector]

networkdefinition ::=
    NETWORK networkname : moduletype
     [ on_block ]
     [ substparamblock ]
    ENDNETWORK

vector ::=    '[' expression ']'
```

```
parexpression ::=
    expression | otherconstvalue

expression    ::=
      expression + expression
    | expression - expression
    | expression * expression
    | expression / expression
    | expression % expression
    | expression ^ expression
    | expression == expression
    | expression != expression
    | expression < expression
    | expression <= expression
    | expression > expression
    | expression >= expression
    | expression ? expression : expression
    | expression AND expression
    | expression OR expression
    | NOT expression
    | '(' expression ')'
    | functionname '(' [ expression ,,, ] ')' $^***$
    | - expression
    | numconstvalue
    | inputvalue
    | [ ANCESTOR ] [ REF ] parametername
    | SIZEOF$^****$ '(' gatename ')'
    | INDEX$^*****$

numconstvalue ::=
    integerconstant | realconstant | timeconstant

otherconstvalue ::=
      'characterconstant'
    | "stringconstant"
    | TRUE
    | FALSE

inputvalue ::=
    INPUT '(' default , "prompt-string" ')'

default ::=
    expression | otherconstvalue
```

* used with distributed execution

** used with the statistical synchronization method

*** max. three arguments. The function name must be declared in the C++ sources with the Define_Function macro.

**** Size of a vector gate.

***** Index in submodule vector.

****** Can appear in any order.

References

[EHW02] K. Entacher, B. Hechenleitner, and S. Wegenkittl. A simple OMNeT++ queuing experiment using parallel streams. *PARALLEL NUMERICS'02 - Theory and Applications*, pages 89–105, 2002. Editors: R. Trobec, P. Zinterhof, M. Vajtersic and A. Uhl.

[EPM99] G. Ewing, K. Pawlikowski, and D. McNickle. Akaroa2: Exploiting network computing by distributing stochastic simulation. In *Proceedings of the European Simulation Multiconference ESM'99, Warsaw, June 1999*, pages 175–181. International Society for Computer Simulation, 1999.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[Hel98] P. Hellekalek. Don't trust parallel Monte Carlo. *ACM SIGSIM Simulation Digest*, 28(1):82–89, jul 1998. Author's page is a great source of information, see http://random.mat.sbg.ac.at/.

[HPvdL95] Jan Heijmans, Alex Paalvast, and Robert van der Leij. Network simulation using the JAR compiler for the OMNeT++ simulation system. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, 1991.

[JC85] Raj Jain and Imrich Chlamtac. The P^2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076–1085, 1985.

[Kof95] Stig Kofoed. Portable multitasking in C++. *Dr. Dobb's Journal*, November 1995. Download sorce from http://www.ddj.com/ftp/1995/1995.11/mtask.zip/.

[Len94] Gábor Lencse. Graphical network editor for OMNeT++. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[LSCK02] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An objected-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002. Source code can be downloaded from http://www.iro.umontreal.ca/ lecuyer/papers.html.

[MN98] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998. Source code can be downloaded from http://www.math.keio.ac.jp/ matumoto/emt.html.

[MvMvdW95] André Maurits, George van Montfort, and Gerard van de Weerd. OMNeT++ extensions and examples. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[PFS86] Bratley P., B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, 1986.

[PJL02] K. Pawlikowski, H. Jeong, and J. Lee. On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine*, pages 132–139, jan 2002.

[Pon91] György Pongor. OMNET: An object-oriented network simulator. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1991.

[Pon92] György Pongor. Statistical Synchronization: A different approach of parallel discrete event simulation. Technical report, University of Technology, Data Communications Laboratory, Lappeenranta, Finland, 1992.

[Pon93] György Pongor. On the efficiency of the Statistical Synchronization Method. In *Proceedings of the European Simulation Symposium (ESS'93), Delft, The Netherlands, Oct. 25-28, 1993*. International Society for Computer Simulation, 1993.

[Var92] András Varga. OMNeT++ - portable simulation environment in C++. In *Proceedings of the Annual Students' Scientific Conference (TDK), 1992*. Technical University of Budapest, 1992. In Hungarian.

[Var94] András Varga. Portable user interface for the OMNeT++ simulation system. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[Var98a] András Varga. K-split – on-line density estimation for simulation result collection. In *Proceedings of the European Simulation Symposium (ESS'98), Nottingham, UK, October 26-28*. International Society for Computer Simulation, 1998.

[Var98b] András Varga. Parameterized topologies for simulation programs. In *Proceedings of the Western Multiconference on Simulation (WMC'98) / Communication Networks and Distributed Systems (CNDS'98), San Diego, CA, January 11-14*. International Society for Computer Simulation, 1998.

[Var99] András Varga. Using the OMNeT++ discrete event simulation system in education. *IEEE Transactions on Education*, 42(4):372, November 1999. on CD-ROM issue; journal contains abstract.

[Vas96] Zoltán Vass. PVM extension of OMNeT++ to support Statistical Synchronization. Master's thesis, Technical University of Budapest, 1996. In Hungarian.

[VF97] András Varga and Babak Fakhamzadeh. The K-Split algorithm for the PDF approximation of multi-dimensional empirical distributions without storing observations. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997*, pages 94–98. International Society for Computer Simulation, 1997.

[VP97] András Varga and György Pongor. Flexible topology description language for simulation programs. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997*, pages 225–229, 1997.

[Wel95] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, 1995.

Index