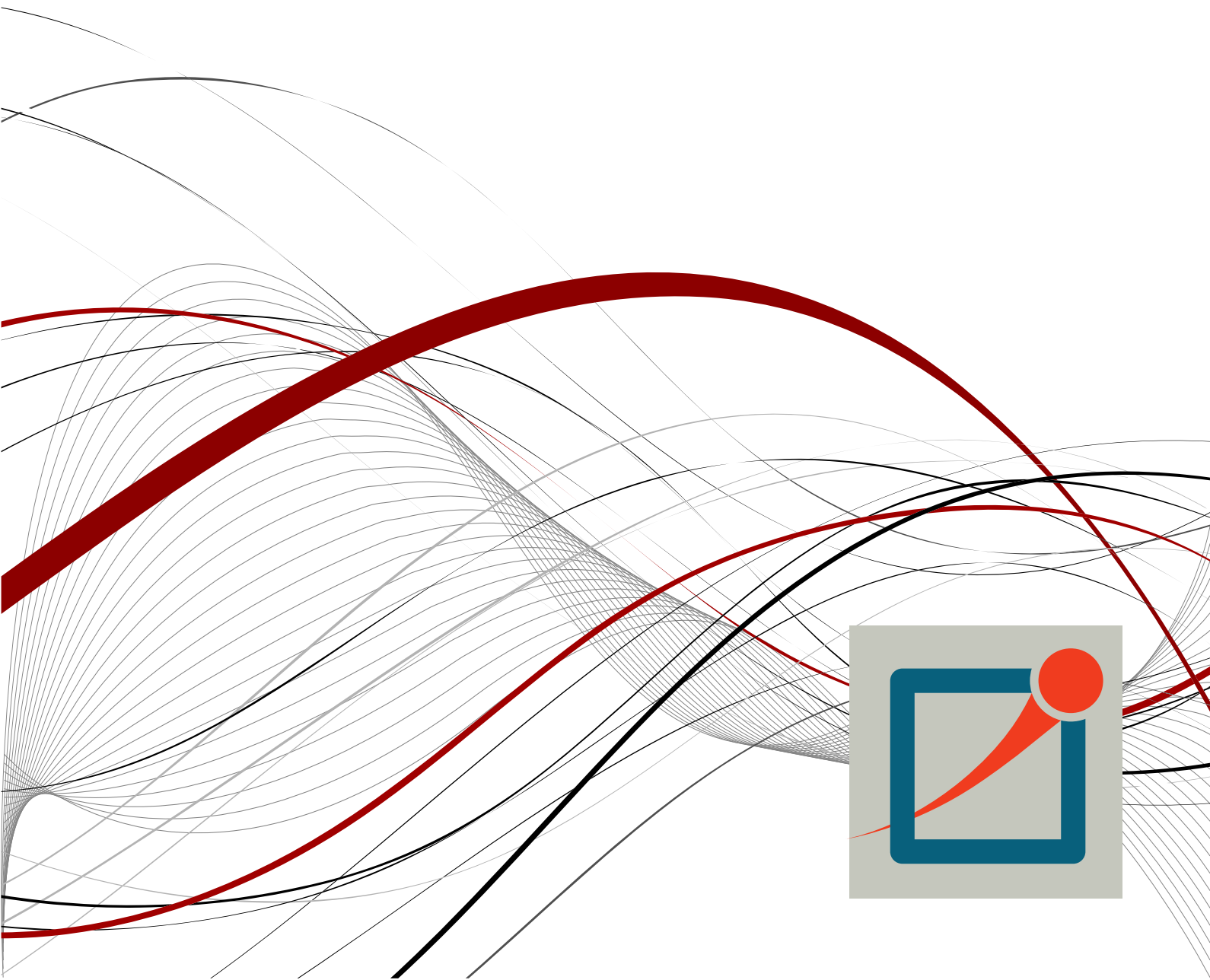
OMNeT++

User Manual

Version 4.5



Chapters

Contents **v**

1 Introduction **1**

2 Overview **3**

3 The NED Language **11**

4 Simple Modules **47**

5 Messages and Packets **119**

6 Message Definitions **129**

7 The Simulation Library **151**

8 Network Graphics And Animation **199**

9 Building Simulation Programs **213**

10 Configuring Simulations **221**

11 Running Simulations **239**

12 Result Recording and Analysis **255**

13 Eventlog **267**

14 Documenting NED and Messages **271**

15 Parallel Distributed Simulation **279**

16 Plug-in Extensions **289**

17 Embedding the Simulation Kernel **297**

A NED Reference **307**

B NED Language Grammar **335**

C NED XML Binding **351**

D NED Functions **359**

E Message Definitions Grammar **365**

F Display String Tags **373**

G Configuration Options **377**

H Result File Formats **389**

I Eventlog File Format **397**

References **403**

Index **406**

Contents

Contents **v**

1 Introduction **1**
1.1 What Is OMNeT++? 1
1.2 Organization of This Manual 2

2 Overview **3**
2.1 Modeling Concepts 3
2.1.1 Hierarchical Modules 4
2.1.2 Module Types 4
2.1.3 Messages, Gates, Links 5
2.1.4 Modeling of Packet Transmissions 5
2.1.5 Parameters 5
2.1.6 Topology Description Method 6
2.2 Programming the Algorithms 6
2.3 Using OMNeT++ 6
2.3.1 Building and Running Simulations 6
2.3.2 What Is in the Distribution 8

3 The NED Language **11**
3.1 NED Overview 11
3.2 NED Quickstart 12
3.2.1 The Network 12
3.2.2 Introducing a Channel 14
3.2.3 The App, Routing, and Queue Simple Modules 14
3.2.4 The Node Compound Module 15
3.2.5 Putting It Together 17
3.3 Simple Modules 17
3.4 Compound Modules 19
3.5 Channels 20

3.6 Parameters . . . 22
3.7 Gates . . . 29
3.8 Submodules . . . 30
3.9 Connections . . . 32
3.10 Multiple Connections . . . 34
3.10.1 Connection Patterns . . . 36
3.11 Parametric Submodule and Connection Types . . . 37
3.11.1 Parametric Submodule Types . . . 37
3.11.2 Parametric Connection Types . . . 39
3.12 Metadata Annotations (Properties) . . . 39
3.13 Inheritance . . . 42
3.14 Packages . . . 42

4 Simple Modules **47**
4.1 Simulation Concepts . . . 47
4.1.1 Discrete Event Simulation . . . 47
4.1.2 The Event Loop . . . 48
4.1.3 Events and Event Execution Order in OMNeT++ . . . 48
4.1.4 Simulation Time . . . 49
4.1.5 FES Implementation . . . 50
4.2 Components, Simple Modules, Channels . . . 50
4.3 Defining Simple Module Types . . . 52
4.3.1 Overview . . . 52
4.3.2 Constructor . . . 53
4.3.3 Initialization and Finalization . . . 53
4.4 Adding Functionality to cSimpleModule . . . 56
4.4.1 handleMessage() . . . 56
4.4.2 activity() . . . 61
4.4.3 How to Avoid Global Variables . . . 66
4.4.4 Reusing Module Code via Subclassing . . . 66
4.5 Accessing Module Parameters . . . 67
4.5.1 Volatile and Non-Volatile Parameters . . . 67
4.5.2 Changing a Parameter's Value . . . 69
4.5.3 Further cPar Methods . . . 69
4.5.4 Emulating Parameter Arrays . . . 70
4.5.5 handleParameterChange() . . . 70
4.6 Accessing Gates and Connections . . . 71
4.6.1 Gate Objects . . . 71
4.6.2 Connections . . . 75

4.6.3 The Connection's Channel . . . 76
4.7 Sending and Receiving Messages . . . 76
4.7.1 Self-Messages . . . 77
4.7.2 Sending Messages . . . 78
4.7.3 Broadcasts and Retransmissions . . . 79
4.7.4 Delayed Sending . . . 80
4.7.5 Direct Message Sending . . . 81
4.7.6 Packet Transmissions . . . 82
4.7.7 Receiving Messages with activity() . . . 84
4.8 Channels . . . 86
4.8.1 Overview . . . 86
4.8.2 The Channel API . . . 86
4.8.3 Channel Examples . . . 88
4.9 Stopping the Simulation . . . 89
4.9.1 Normal Termination . . . 89
4.9.2 Raising Errors . . . 89
4.10 Finite State Machines . . . 90
4.11 Navigating the Module Hierarchy . . . 94
4.12 Direct Method Calls Between Modules . . . 96
4.13 Dynamic Module Creation . . . 97
4.13.1 When Do You Need Dynamic Module Creation . . . 97
4.13.2 Overview . . . 97
4.13.3 Creating Modules . . . 98
4.13.4 Deleting Modules . . . 99
4.13.5 Module Deletion and finish() . . . 99
4.13.6 Creating Connections . . . 100
4.13.7 Removing Connections . . . 101
4.14 Signals . . . 101
4.14.1 Design Considerations and Rationale . . . 102
4.14.2 The Signals Mechanism . . . 102
4.14.3 Listening to Model Changes . . . 107
4.15 Signal-Based Statistics Recording . . . 109
4.15.1 Motivation . . . 109
4.15.2 Declaring Statistics . . . 110
4.15.3 Statistics Recording for Dynamically Registered Signals . . . 114
4.15.4 Adding Result Filters and Recorders Programmatically . . . 115
4.15.5 Emitting Signals . . . 116
4.15.6 Writing Result Filters and Recorders . . . 117

5 Messages and Packets **119**
5.1 Overview . . . 119
5.2 The cMessage Class . . . 120
5.2.1 Basic Usage . . . 120
5.2.2 Duplicating Messages . . . 121
5.2.3 Message IDs . . . 122
5.2.4 Control Info . . . 122
5.2.5 Information About the Last Arrival . . . 122
5.2.6 Display String . . . 123
5.3 Self-Messages . . . 123
5.3.1 Using a Message as Self-Message . . . 123
5.3.2 Context Pointer . . . 124
5.4 The cPacket Class . . . 124
5.4.1 Basic Usage . . . 124
5.4.2 Identifying the Protocol . . . 124
5.4.3 Information About the Last Transmission . . . 125
5.4.4 Encapsulating Packets . . . 125
5.4.5 Reference Counting . . . 126
5.4.6 Encapsulating Several Packets . . . 126
5.5 Attaching Parameters and Objects . . . 127
5.5.1 Attaching Objects . . . 127
5.5.2 Attaching Parameters . . . 127

6 Message Definitions **129**
6.1 Introduction . . . 129
6.1.1 The First Message Class . . . 129
6.2 Messages and Packets . . . 130
6.2.1 Defining Messages and Packets . . . 130
6.2.2 Field Data Types . . . 131
6.2.3 Initial Values . . . 132
6.2.4 Enums . . . 132
6.2.5 Fixed-Size Arrays . . . 133
6.2.6 Variable-Size Arrays . . . 133
6.2.7 Classes and Structs as Fields . . . 134
6.2.8 Pointer Fields . . . 134
6.2.9 Inheritance . . . 135
6.2.10 Assignment of Inherited Fields . . . 135
6.3 Classes . . . 136
6.4 Structs . . . 136

6.5 Literal C++ Blocks . . . 137
6.6 Using C++ Types . . . 138
6.6.1 Announcing Types to the Message Compiler . . . 138
6.6.2 Making the C++ Declarations Available . . . 139
6.6.3 Putting it Together . . . 139
6.7 Customizing the Generated Class . . . 140
6.7.1 Customizing Method Names . . . 140
6.7.2 Customizing the Class via Inheritance . . . 141
6.7.3 Abstract Fields . . . 142
6.8 Using Standard Container Classes for Fields . . . 143
6.8.1 Typedefs . . . 143
6.8.2 Abstract Fields . . . 144
6.9 Namespaces . . . 146
6.9.1 Declaring a Namespace . . . 146
6.9.2 C++ Blocks and Namespace . . . 146
6.9.3 Type Announcements and Namespace . . . 147
6.10 Descriptor Classes . . . 148
6.11 Summary . . . 149

7 The Simulation Library 151
7.1 Class Library Conventions . . . 152
7.1.1 Base Class . . . 152
7.1.2 Setting and Getting Attributes . . . 152
7.1.3 getClassName() . . . 152
7.1.4 Object Names . . . 152
7.1.5 Object Full Name and Full Path . . . 153
7.1.6 Copying and Duplicating Objects . . . 154
7.1.7 Iterators . . . 154
7.1.8 Error Handling . . . 154
7.2 Logging from Modules . . . 155
7.3 Simulation Time Conversion . . . 155
7.4 Generating Random Numbers . . . 156
7.4.1 Random Number Generators . . . 156
7.4.2 Random Number Streams, RNG Mapping . . . 157
7.4.3 Accessing The RNGs . . . 157
7.4.4 Random Variates . . . 158
7.4.5 Random Numbers from Histograms . . . 159
7.5 Container Classes . . . 159
7.5.1 Queue class: cQueue . . . 159

7.5.2 Expandable Array: cArray . . . 161
7.6 Routing Support: cTopology . . . 162
7.6.1 Overview . . . 162
7.6.2 Basic Usage . . . 163
7.6.3 Shortest Paths . . . 164
7.7 Pattern Matching . . . 166
7.7.1 cPatternMatcher . . . 166
7.7.2 cMatchExpression . . . 168
7.8 Statistics and Distribution Estimation . . . 169
7.8.1 cStatistic and Descendants . . . 169
7.8.2 Distribution Estimation . . . 170
7.8.3 The k-split Algorithm . . . 173
7.8.4 Transient Detection and Result Accuracy . . . 176
7.9 Recording Simulation Results . . . 177
7.9.1 Output Vectors: cOutVector . . . 177
7.9.2 Output Scalars . . . 178
7.10 Watches and Snapshots . . . 179
7.10.1 Basic Watches . . . 179
7.10.2 Read-write Watches . . . 180
7.10.3 Structured Watches . . . 180
7.10.4 STL Watches . . . 180
7.10.5 Snapshots . . . 181
7.10.6 Getting Coroutine Stack Usage . . . 182
7.11 Defining New NED Functions . . . 183
7.11.1 Define_NED_Function() . . . 183
7.11.2 Define_NED_Math_Function() . . . 188
7.12 Deriving New Classes . . . 189
7.12.1 cOwnedObject or Not? . . . 189
7.12.2 cOwnedObject Virtual Methods . . . 189
7.12.3 Class Registration . . . 190
7.12.4 Details . . . 191
7.13 Object Ownership Management . . . 194
7.13.1 The Ownership Tree . . . 194
7.13.2 Managing Ownership . . . 195

8 Network Graphics And Animation 199

8.1 Display Strings . . . 199
8.1.1 Display String Syntax . . . 199
8.1.2 Display String Placement . . . 199

8.1.3 Display String Inheritance . . . 200
8.1.4 Display String Tags Used in Submodule Context . . . 201
8.1.5 Display String Tags Used in Module Background Context . . . 205
8.1.6 Connection Display Strings . . . 206
8.1.7 Message Display Strings . . . 207
8.2 Parameter Substitution . . . 208
8.3 Colors . . . 208
8.3.1 Color Names . . . 208
8.3.2 Icon Colorization . . . 208
8.4 Icons . . . 209
8.4.1 The Image Path . . . 209
8.4.2 Categorized Icons . . . 209
8.4.3 Icon Size . . . 210
8.5 Layouting . . . 210
8.6 Enhancing Animation . . . 211
8.6.1 Changing Display Strings at Runtime . . . 211
8.6.2 Bubbles . . . 212

9 Building Simulation Programs 213
9.1 Overview . . . 213
9.2 Using gcc . . . 215
9.2.1 The opp_makemake Tool . . . 215
9.2.2 Basic Use . . . 215
9.2.3 Debug and Release Builds . . . 216
9.2.4 Debugging the Makefile . . . 216
9.2.5 Using External C/C++ Libraries . . . 216
9.2.6 Building Directory Trees . . . 217
9.2.7 Automatic Include Dirs . . . 217
9.2.8 Dependency Handling . . . 217
9.2.9 Out-of-Directory Build . . . 217
9.2.10 Building Shared and Static Libraries . . . 218
9.2.11 Recursive Builds . . . 218
9.2.12 Customizing the Makefile . . . 219
9.2.13 Projects with Multiple Source Trees . . . 219
9.2.14 A Multi-Directory Example . . . 219

10 Configuring Simulations 221
10.1 The Configuration File . . . 221
10.1.1 An Example . . . 221

10.1.2 File Syntax . . . 222
10.1.3 File Inclusion . . . 223
10.2 Sections . . . 223
10.2.1 The [General] Section . . . 223
10.2.2 Named Configurations . . . 224
10.2.3 Section Inheritance . . . 224
10.3 Assigning Module Parameters . . . 226
10.3.1 Using Wildcard Patterns . . . 226
10.3.2 Using the Default Values . . . 227
10.4 Parameter Studies . . . 228
10.4.1 Iterations . . . 230
10.4.2 Named Iteration Variables . . . 230
10.4.3 Parallel Iteration . . . 232
10.4.4 Predefined Variables, Run ID . . . 232
10.4.5 Constraint Expression . . . 233
10.4.6 Repeating Runs with Different Seeds . . . 233
10.4.7 Experiment-Measurement-Replication . . . 234
10.5 Configuring the Random Number Generators . . . 236
10.5.1 Number of RNGs . . . 236
10.5.2 RNG Choice . . . 236
10.5.3 RNG Mapping . . . 237
10.5.4 Automatic Seed Selection . . . 237
10.5.5 Manual Seed Configuration . . . 238

11 Running Simulations 239
11.1 Introduction . . . 239
11.1.1 Running a Simulation Executable . . . 239
11.1.2 Running a Shared Library . . . 242
11.1.3 Controlling the Run . . . 242
11.2 Cmdenv: the Command-Line Interface . . . 243
11.2.1 Example Run . . . 243
11.2.2 Command-Line Options . . . 244
11.2.3 Cmdenv Ini File Options . . . 244
11.2.4 Interpreting Cmdenv Output . . . 244
11.3 Tkenv: the Graphical User Interface . . . 246
11.3.1 Command-Line and Configuration Options . . . 246
11.4 Batch Execution . . . 246
11.4.1 Using Cmdenv . . . 247
11.4.2 Using Shell Scripts . . . 247

11.4.3 Using opp_runall . . . 248
11.5 Akaroa Support: Multiple Replications in Parallel . . . 248
11.5.1 Introduction . . . 248
11.5.2 What Is Akaroa . . . 249
11.5.3 Using Akaroa with OMNeT++ . . . 249
11.6 Troubleshooting . . . 251
11.6.1 Unrecognized Configuration Option . . . 251
11.6.2 Stack Problems . . . 251
11.6.3 Memory Leaks and Crashes . . . 253
11.6.4 Simulation Executes Slowly . . . 254

12 Result Recording and Analysis 255
12.1 Result Recording . . . 255
12.1.1 Using Signals and Declared Statistics . . . 255
12.1.2 Direct Result Recording . . . 256
12.2 Configuring Result Collection . . . 256
12.2.1 Configuring Signal-Based Statistics Recording . . . 256
12.2.2 Warm-up Period . . . 257
12.2.3 Result File Names . . . 258
12.2.4 Configuring Scalar Results . . . 258
12.2.5 Configuring Output Vectors . . . 259
12.2.6 Saving Parameters as Scalars . . . 259
12.2.7 Recording Precision . . . 260
12.3 Overview of the Result File Formats . . . 261
12.3.1 Output Vector Files . . . 261
12.3.2 Scalar Result Files . . . 262
12.4 The Analysis Tool in the Simulation IDE . . . 262
12.5 Scave Tool . . . 263
12.5.1 The *filter* Command . . . 263
12.5.2 The *index* Command . . . 264
12.5.3 The *summary* Command . . . 264
12.6 Alternative Statistical Analysis and Plotting Tools . . . 264
12.6.1 GNU R . . . 264
12.6.2 NumPy, SciPy and MatPlotLib . . . 265
12.6.3 MATLAB or Octave . . . 265
12.6.4 Gnuplot . . . 265
12.6.5 ROOT . . . 265
12.6.6 Grace . . . 266
12.6.7 Spreadsheet Programs . . . 266

13 Eventlog **267**
13.1 Introduction . . . 267
13.2 Configuration . . . 267
13.2.1 File Name . . . 268
13.2.2 Recording Intervals . . . 268
13.2.3 Recording Modules . . . 268
13.2.4 Recording Message Data . . . 268
13.3 Eventlog Tool . . . 269
13.3.1 Filter . . . 269
13.3.2 Echo . . . 269

14 Documenting NED and Messages **271**
14.1 Overview . . . 271
14.2 Documentation Comments . . . 271
14.2.1 Private Comments . . . 272
14.2.2 More on Comment Placement . . . 272
14.3 Referring to Other NED and Message Types . . . 273
14.3.1 Automatic Linking . . . 273
14.3.2 Tilde Linking . . . 274
14.4 Text Layout and Formatting . . . 274
14.4.1 Paragraphs and Lists . . . 274
14.4.2 Special Tags . . . 274
14.4.3 Text Formatting Using HTML . . . 275
14.4.4 Escaping HTML Tags . . . 276
14.5 Customizing and Adding Pages . . . 277
14.5.1 Adding a Custom Title Page . . . 277
14.5.2 Adding Extra Pages . . . 277
14.5.3 Incorporating Externally Created Pages . . . 278
14.6 File inclusion . . . 278

15 Parallel Distributed Simulation **279**
15.1 Introduction to Parallel Discrete Event Simulation . . . 279
15.2 Assessing Available Parallelism in a Simulation Model . . . 280
15.3 Parallel Distributed Simulation Support in OMNeT++ . . . 281
15.3.1 Overview . . . 281
15.3.2 Parallel Simulation Example . . . 282
15.3.3 Placeholder Modules, Proxy Gates . . . 283
15.3.4 Configuration . . . 284
15.3.5 Design of PDES Support in OMNeT++ . . . 286

16 Plug-in Extensions **289**
16.1 Overview . . . 289
16.2 Plug-in Descriptions . . . 290
16.2.1 Defining a New Random Number Generator . . . 290
16.2.2 Defining a New Scheduler . . . 290
16.2.3 Defining a New Configuration Provider . . . 290
16.2.4 Defining a New Output Scalar Manager . . . 292
16.2.5 Defining a New Output Vector Manager . . . 292
16.2.6 Defining a New Snapshot Manager . . . 292
16.3 Accessing the Configuration . . . 292
16.3.1 Defining New Configuration Options . . . 292
16.3.2 Reading Values from the Configuration . . . 293
16.4 Implementing a New User Interface . . . 294

17 Embedding the Simulation Kernel **297**
17.1 Architecture . . . 297
17.2 Embedding the OMNeT++ Simulation Kernel . . . 298
17.2.1 The main() Function . . . 299
17.2.2 The simulate() Function . . . 300
17.2.3 Providing an Environment Object . . . 301
17.2.4 Providing a Configuration Object . . . 302
17.2.5 Loading NED Files . . . 303
17.2.6 How to Eliminate NED Files . . . 303
17.2.7 Assigning Module Parameters . . . 303
17.2.8 Extracting Statistics from the Model . . . 304
17.2.9 The Simulation Loop . . . 305
17.2.10 Multiple, Coexisting Simulations . . . 306
17.2.11 Installing a Custom Scheduler . . . 306
17.2.12 Multi-Threaded Programs . . . 306

A NED Reference **307**
A.1 Syntax . . . 307
A.1.1 NED File Name Extension . . . 307
A.1.2 NED File Encoding . . . 307
A.1.3 Reserved Words . . . 307
A.1.4 Identifiers . . . 308
A.1.5 Case Sensitivity . . . 308
A.1.6 Literals . . . 308
A.1.7 Comments . . . 308

A.1.8 Grammar . . . 308
A.2 Built-in Definitions . . . 309
A.3 Packages . . . 310
A.3.1 Package Declaration . . . 310
A.3.2 Directory Structure, package.ned . . . 310
A.4 Components . . . 311
A.4.1 Simple Modules . . . 311
A.4.2 Compound Modules . . . 311
A.4.3 Networks . . . 311
A.4.4 Channels . . . 312
A.4.5 Module Interfaces . . . 312
A.4.6 Channel Interfaces . . . 312
A.4.7 Resolving the C++ Implementation Class . . . 313
A.4.8 Properties . . . 313
A.4.9 Parameters . . . 315
A.4.10 Pattern Assignments . . . 316
A.4.11 Gates . . . 317
A.4.12 Submodules . . . 318
A.4.13 Connections . . . 319
A.4.14 Conditional and Loop Connections, Connection Groups . . . 323
A.4.15 Inner Types . . . 323
A.4.16 Name Uniqueness . . . 323
A.4.17 Parameter Assignment Order . . . 324
A.4.18 Type Name Resolution . . . 326
A.4.19 Resolution of Parametric Types . . . 326
A.4.20 Implementing an Interface . . . 329
A.4.21 Inheritance . . . 329
A.4.22 Network Build Order . . . 330
A.5 Expressions . . . 331
A.5.1 Operators . . . 331
A.5.2 Referencing Parameters and Loop Variables . . . 332
A.5.3 The `index` Operator . . . 332
A.5.4 The `sizeof()` Operator . . . 332
A.5.5 Functions . . . 332
A.5.6 Units of Measurement . . . 333

B NED Language Grammar **335**

C NED XML Binding **351**

D NED Functions **359**

E Message Definitions Grammar **365**

F Display String Tags **373**
F.1 Module and Connection Display String Tags 373
F.2 Message Display String Tags 375

G Configuration Options **377**
G.1 Configuration Options 377
G.2 Predefined Configuration Variables 386

H Result File Formats **389**
H.1 Version 390
H.2 Run Declaration 390
H.3 Attributes 391
H.4 Module Parameters 391
H.5 Scalar Data 392
H.6 Vector Declaration 392
H.7 Vector Data 393
H.8 Index Header 393
H.9 Index Data 393
H.10 Statistics Object 394
H.11 Field 395
H.12 Histogram Bin 395

I Eventlog File Format **397**
I.1 Supported Entry Types and Their Attributes 398

References **403**

Index **406**

Chapter 1

Introduction

1.1 What Is OMNeT++?

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

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

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

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

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

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

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

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

1.2 Organization of This Manual

The manual is organized as follows:

- The Chapters 1 and 2 contain introductory material
- The second group of chapters, 3, 4 and 7 are the programming guide. They present the NED language, describe the simulation concepts and their implementation in OMNeT++, explain how to write simple modules, and describe the class library.
- The chapters 8 and 14 explain how to customize the network graphics and how to write NED source code comments from which documentation can be generated.
- Chapters 9, 10, 11 and 12 deal with practical issues like building and running simulations and analyzing results, and describe the tools OMNeT++ provides to support these tasks.
- Chapter 15 is devoted to the support of distributed execution.
- Chapters 16 and 17 explain the architecture and internals of OMNeT++, as well as ways to extend it and embed it into larger applications.
- The appendices provide a reference on the NED language, configuration options, file formats, and other details.

Chapter 2

Overview

2.1 Modeling Concepts

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

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

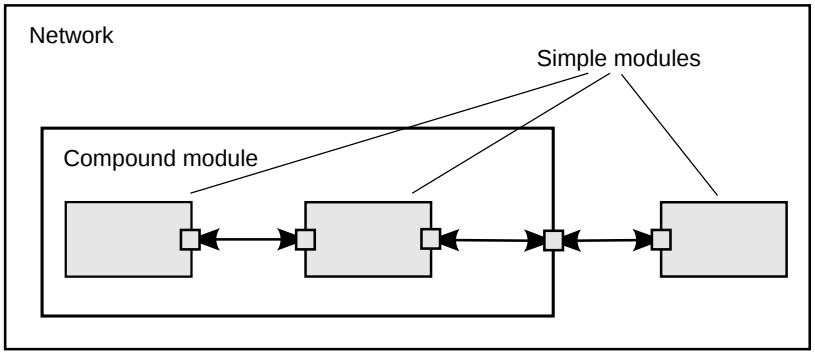


Figure 2.1: Simple and compound modules

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

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

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

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

2.1.1 Hierarchical Modules

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

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

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

2.1.2 Module Types

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

When a module type is used as a building block, it makes no difference whether it is a simple or compound module. This allows the user to split a simple module into several simple modules embedded into a compound module, or vice versa, to aggregate the functionality of a compound module into a single simple module, without affecting existing users of the module type.

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

2.1.3 Messages, Gates, Links

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

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

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

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

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

2.1.4 Modeling of Packet Transmissions

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

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

2.1.5 Parameters

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

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

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

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

2.1.6 Topology Description Method

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

2.2 Programming the Algorithms

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

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

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

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

2.3 Using OMNeT++

2.3.1 Building and Running Simulations

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

An OMNeT++ model consists of the following parts:

- NED language topology description(s) (`.ned` files) that describe the module structure with parameters, gates, etc. NED files can be written using any text editor, but the OMNeT++ IDE provides excellent support for two-way graphical and text editing.

- Message definitions (`.msg` files). You can define various message types and add data fields to them. OMNeT++ will translate message definitions into full-fledged C++ classes.

- Simple module sources. They are C++ files, with `.h`/`.cc` suffix.

The simulation system provides the following components:

- Simulation kernel. This contains the code that manages the simulation and the simulation class library. It is written in C++, compiled into a shared or static library.

- User interfaces. OMNeT++ user interfaces are used in simulation execution, to facilitate debugging, demonstration, or batch execution of simulations. They are written in C++, compiled into libraries.

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

Running the Simulation and Analyzing the Results

The simulation may be compiled as a standalone program executable; thus it can be run on other machines without OMNeT++ being present, or it can be created as a shared library. In this case the OMNeT++ shared libraries must be present on that system. When the program is started, it first reads all NED files containing your model topology, then it reads a configuration file (usually called `omnetpp.ini`). This file contains settings that control how the simulation is executed, values for model parameters, etc. The configuration file can also prescribe several simulation runs; in the simplest case, they will be executed by the simulation program one after another.

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

User Interfaces

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

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

Component Libraries

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

Universal Standalone Simulation Programs

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

2.3.2 What Is in the Distribution

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

The simulation system itself:

```
omnetpp/          OMNeT++ root directory
  bin/            OMNeT++ executables
  include/        header files for simulation models
  lib/            library files
  images/         icons and backgrounds for network graphics
  doc/            manuals, readme files, license, APIs, etc.
    ide-customization-guide/ how to write new wizards for the IDE
    ide-developersguide/ writing extensions for the IDE
    manual/       manual in HTML
    migration/    how to migrate your models from 3.x to 4.0 version
    ned2/         DTD definition of the XML syntax for NED files
    tictoc-tutorial/  introduction into using OMNeT++
    api/          API reference in HTML
    nedxml-api/   API reference for the NEDXML library
    parsim-api/   API reference for the parallel simulation library
  migrate/        tools to help model migration from 3.x to 4.0 version
  src/            OMNeT++ sources
    sim/          simulation kernel
      parsim/     files for distributed execution
      netbuilder/files for dynamically reading NED files
    envir/        common code for user interfaces
    cmdenv/       command-line user interface
    tkenv/        Tcl/Tk-based user interface
    nedxml/       NEDXML library, nedtool, opp_msgc
    scave/        result analysis library
    eventlog/     eventlog processing library
    layout/       graph layouter for network graphics
    common/       common library
    utils/        opp_makemake, opp_test, etc.
```

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

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

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

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

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

Sample simulations are in the `samples` directory.

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

The `contrib` directory contains material from the OMNeT++ community.

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

Chapter 3

The NED Language

3.1 NED Overview

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2 NED Quickstart

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

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

3.2.1 The Network

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

Let the network topology be as in Figure 3.1.

The corresponding NED description would look like this:

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

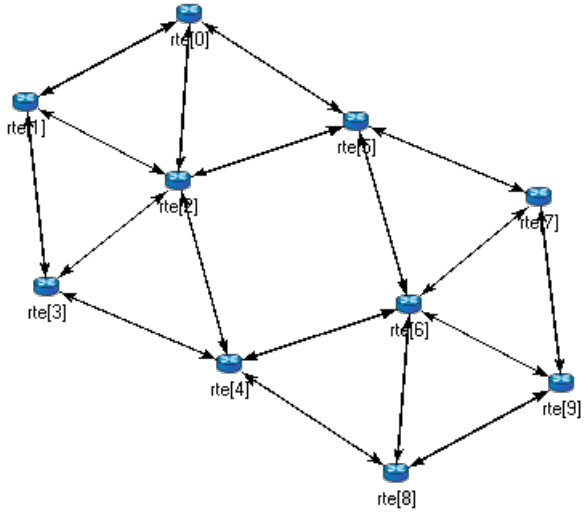


Figure 3.1: The network

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

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

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

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

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

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

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

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

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

3.2.2 Introducing a Channel

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

The improved network will look like this:

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

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

3.2.3 The App, Routing, and Queue Simple Modules

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

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

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

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

simple Routing
{
    ...
}

simple Queue
{
    ...
}
```

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

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

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

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

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

3.2.4 The Node Compound Module

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

```
module Node
{
```

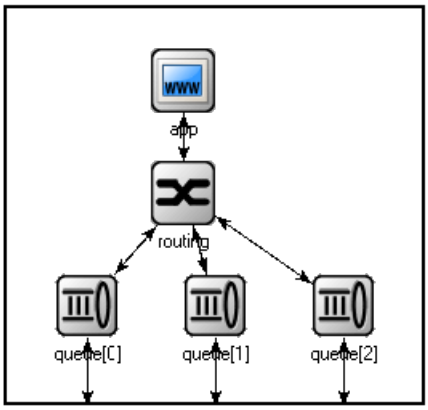


Figure 3.2: The Node compound module

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

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

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

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

3.2.5 Putting It Together

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

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

* * *

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

3.3 Simple Modules

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

An example simple module:

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

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

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

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

```
simple Queue
{
    parameters:
        int capacity;
```

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

If you have several modules that are all in a common namespace, then a better alternative to `@class` is the `@namespace` property. The C++ namespace given with `@namespace` will be prepended to the normal class name. In the following example, the C++ classes will be `mylib::App`, `mylib::Router` and `mylib::Queue`:

```
@namespace(mylib);

simple App {
   ...
}

simple Router {
   ...
}

simple Queue {
   ...
}
```

As you've seen, `@namespace` can be specified at the file level. Moreover, when placed in a file called `package.ned`, the namespace will apply to all files in the same directory and all directories below.

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

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

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

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

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

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

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

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

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

Inheritance in general will be discussed in section 3.13.

3.4 Compound Modules

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

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

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

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

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

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

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

Compound modules may be extended via subclassing. Inheritance may add new submodules and new connections as well, not only parameters and gates. Also, one may refer to

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

inherited submodules, to inherited types etc. What is not possible is to "de-inherit" or modify submodules or connections.

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

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

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

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

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

3.5 Channels

Channels encapsulate parameters and behaviour associated with connections. Channels are like simple modules, in the sense that there are C++ classes behind them. The rules for

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

finding the C++ class for a NED channel type is the same as with simple modules: the default class name is the NED type name unless there is a `@class` property (`@namespace` is also recognized), and the C++ class is inherited when the channel is subclassed.

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

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

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

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

`DelayChannel` has two parameters:

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

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

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

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

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

```
channel Ethernet100 extends ned.DatarateChannel
{
```

```
    datarate = 100Mbps;
    delay = 100us;
    ber = 1e-10;
}
```

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

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

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

Parameters are primarily useful as input to the underlying C++ class, but even if you reuse the underlying C++ class of built-in channel types, they may be read and used by other parts of the model. For example, adding a `cost` parameter (or `@cost` property) may be observed by the routing algorithm and used for routing decisions. The following example shows a `cost` parameter, and annotation using a property (`@backbone`).

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

3.6 Parameters

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

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

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

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

```
simple App
{
    parameters:
        string protocol;        // protocol to use: "UDP" / "IP" / "ICMP" / ...
        int destAddress;        // destination address
        volatile double sendInterval @unit(s) = default(exponential(1s));
```

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

Assigning a Value

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

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

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

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

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

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

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

It is not only possible to set a parameter from the compound module that contains the submodule, but also from modules higher up in the module tree. If you had a network that employed several `Host` modules, it could be defined like this:

```
network Network
```

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

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

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

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

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

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

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

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

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

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

There is one more wildcard that can be used in pattern assignments, and this is the double asterisk; it matches any sequence of characters including dots, so it can match multiple path elements. An example:

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

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

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

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

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

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

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

Or if you are sure that you don't accidentally assign some other `sendInterval` parameter, you can just write

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

Parameter assignments in the configuration are described in section 10.3.

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

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

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

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

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

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

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

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

Expressions

Parameter values may be given with expressions. NED language expressions have a C-like syntax, with some variations on operator names: binary and logical XOR are `#` and `##`, while `^` has been reassigned to *power-of* instead. The `+` operator does string concatenation as well as numeric addition. Expressions can use various numeric, string, stochastic and other functions (`fabs()`, `toUpper()`, `uniform()`, `erlang_k()`, etc.).

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

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

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

volatile

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

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

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

Because of the **`volatile`** modifier, the queue module's C++ implementation is expected to re-read the `serviceTime` parameter whenever a value is needed; that is, for every job serviced. Thus, if `serviceTime` is assigned an expression like `uniform(0.5s, 1.5s)`, every job will have a different, random service time. To highlight this effect, here's how you can have a time-varying parameter by exploiting the `simTime()` NED function that returns the current simulation time:

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

In practice, a volatile parameters are typically used as a configurable source of random numbers for modules.

NOTE: This does not mean that a non-volatile parameter could not be assigned a random value like `uniform(0.5s, 1.5s)`. It can, but that would have a totally different effect: the simulation would use a constant service time, say `1.2975367s`, chosen randomly at the beginning of the simulation.

Units

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

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

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

NOTE: The list of units accepted by OMNeT++ is listed in the Appendix, see A.5.6. Unknown units (`bogomips`, etc.) can also be used, but there are no conversions for them, i.e. decimal prefixes will not be recognized.

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

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

XML Parameters

Sometimes modules need complex data structures as input, which is something that cannot be done well with module parameters. One solution is to place the input data into a custom configuration file, pass the file name to the module in a string parameter, and let the module read and parse the file.

It is somewhat easier if the configuration uses XML syntax, because OMNeT++ contains built-in support for XML files. Using an XML parser (LibXML2 or Expat), OMNeT++ reads and DTD-validates the file (if the XML document contains a DOCTYPE), caches the file (so that references to it from several modules will result in the file being loaded only once), allows selection of parts of the document using an XPath-subset notation, and presents the contents in a DOM-like object tree.

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

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

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

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

It is also possible to assign an XML element within a file to the parameter, which is useful if you want to group the input of several modules into a single XML file. For example, the following XML file contains two profiles with the IDs *gen1* and *gen2*:

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

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

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

It is also possible to create an XML document from a string constant, using the **xml()** function. This is especially useful for creating a default value for **xml** parameters. An example:

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

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

3.7 Gates

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

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

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

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

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

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

Let us see some examples.

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

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

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

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

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

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

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

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

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

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

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

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

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

3.8 Submodules

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

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

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

The basic syntax of submodules is shown below:

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

```
        ...
}
```

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

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

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

An empty body may be omitted, that is,

```
      queue: Queue;
```

is the same as

```
      queue: Queue {
      }
```

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

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

The condition is less useful with submodule vectors, as one could also use a zero vector size.

3.9 Connections

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

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

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

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

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

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

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

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

Channel Specification

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

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

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

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

```
a.g++ <--> {delay=10ms;} <--> b.g++;
a.g++ <--> {delay=10ms; ber=1e-8;} <--> b.g++;
```

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

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

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

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

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

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

Also, it is not always easy to figure out which gate indices map to the connections you want to configure. If connection objects could be given names to override the default name "`channel`", that would make it easier to identify connections in patterns. This feature is planned for future OMNeT++ releases.

Channel Names

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

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

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

Examples:

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

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

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

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

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

Then consider the following inifile:

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

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

3.10 Multiple Connections

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

This will be shown in the following examples.

Chain

One can create a chain of modules like this:

```
module Chain
    parameters:
        int count;
    submodules:
        node[count] : Node {
            gates:
```

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

Binary Tree

One can build a binary tree in the following way:

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

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

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

Random Graph

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

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

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

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

3.10.1 Connection Patterns

Several approaches can be used when you want to create complex topologies which have a regular structure; three of them are described below.

"Subgraph of a Full Graph"

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

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

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

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

"Connections of Each Node"

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

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

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

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

"Enumerate All Connections"

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

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

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

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

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

3.11 Parametric Submodule and Connection Types

3.11.1 Parametric Submodule Types

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

Let us begin with an example:

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

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

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

```
network Net6
{
    parameters:
        string nodeTypePrefix;
        int variant;
    submodules:
        node[6]: <nodeTypePrefix + "Node" + string(variant)> like INode {
            ...
}
```

The corresponding NED declarations:

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

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

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

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

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

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

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

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

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

```
        ...
}
```

3.11.2 Parametric Connection Types

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

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

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

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

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

A default value may also be given:

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

The corresponding type pattern assignments:

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

3.12 Metadata Annotations (Properties)

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

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

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

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

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

module Example
{
    parameters:
       @node;   // module property
```

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

Property Indices

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

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

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

This example declares two statistics as `@statistic` properties, `@statistic[eed]` and `@statistic[jitter]`. Property values within the parentheses are used to supply additional info, like a more descriptive name (`title="..."` or a unit (`unit=s`). Property indices can be conveniently accessed from the C++ API as well; for example it is possible to ask what indices exist for the `"statistic"` property, and it will return a list containing `"eed"` and `"jitter"`).

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

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

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

Indices also make it possible to override entries via inheritance:

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

Data Model

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

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

Properties may contain lists:

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

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

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

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

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

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

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

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

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

Overriding and Extending Property Values

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

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

Some examples:

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

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

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

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

3.13 Inheritance

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

In NED, a type may only extend (**`extends`** keyword) an element of the same component type: a simple module may only extend a simple module, compound module may only extend a compound module, and so on. Single inheritance is supported for modules and channels, and multiple inheritance is supported for module interfaces and channel interfaces. A network is a shorthand for a compound module with the `@isNetwork` property set, so the same rules apply to it as to compound modules.

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

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

Inheritance may:

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

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

3.14 Packages

Having all NED files in a single directory is fine for small simulation projects. When a project grows, however, it sooner or later becomes necessary to introduce a directory structure, and

sort the NED files into them. NED natively supports directory trees with NED files, and calls directories *packages*. Packages are also useful for reducing name conflicts, because names can be qualified with the package name.

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

Overview

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

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

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

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

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

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

The toplevel `package.ned` file can be used to designate the root package, which is useful for eliminating a few levels of empty directories resulting from the package naming convention. For example, if you have a project where you want to have all NED types under the `org.example.myproject` package but don't want to have the directories named `org`, `example` and `myproject` in the source tree, then you can put a `package.ned` file in the source root directory with the package declaration `org.example.myproject`. This will cause a directory `foo` under the root to be interpreted as package `org.example.myproject.foo`, and NED files in them must contain that as package declaration. Only the root `package.ned` can define the package, `package.ned` files in subdirectories must follow it.

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

```
INET/
    src/
        base/
        transport/
            tcp/
```

```
            udp/
            ...
        networklayer/
        linklayer/
        ...
    examples/
        adhoc/
        ethernet/
        ...
```

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

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

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

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

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

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

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

Name Resolution, Imports

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

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

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

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

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

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

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

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

Name Resolution With "like"

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

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

Let us see the following example:

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

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

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

The Default Package

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

Chapter 4

Simple Modules

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

4.1 Simulation Concepts

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

4.1.1 Discrete Event Simulation

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

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

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

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

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

4.1.2 The Event Loop

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

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

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

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

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

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

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

4.1.3 Events and Event Execution Order in OMNeT++

OMNeT++ uses messages to represent events. Each event is represented by an instance of the `cMessage` class or one its subclasses; there is no separate event class. Messages are sent from one module to another – this means that the place where the "event will occur" is the *message's destination module*, and the model time when the event occurs is the *arrival time* of the message. Events like "timeout expired" are implemented by the module sending a message to itself.

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

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

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

4.1.4 Simulation Time

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

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

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

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

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

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

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

Other useful methods of `SimTime` include `str()`, which returns the value as a string; `parse()`, which converts a string to `SimTime`; `raw()`, which returns the underlying `int64` value; `getScaleExp()`, which returns the global scale exponent; and `getMaxTime`, which returns the maximum simulation time that can be represented at the current scale exponent.

Compatibility

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

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

NOTE: Why did OMNeT++ switch to `int64`-based simulation time? `double`'s mantissa is only 52 bits long, and this caused problems in long simulations that relied on fine-grained timing, for example MAC protocols. Other problems were the accumulation of rounding errors, and non-associativity (often $(x+y)+z \neq x+(y+z)$, see [Gol91]) which meant that two `double` simulation times could not be reliably compared for equality.

4.1.5 FES Implementation

The implementation of the FES is a crucial factor in the performance of a discrete event simulator. In OMNeT++, the FES is implemented with *binary heap*, the most widely used data structure for this purpose. Heap is generally considered the best algorithm, although exotic data structures like *skiplist* may perform better than heap in some cases. In case you are interested, the FES implementation is in the `cMessageHeap` class, but knowledge of the FES implementation is not necessary for the typical simulation programmer.

4.2 Components, Simple Modules, Channels

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

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

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

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

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

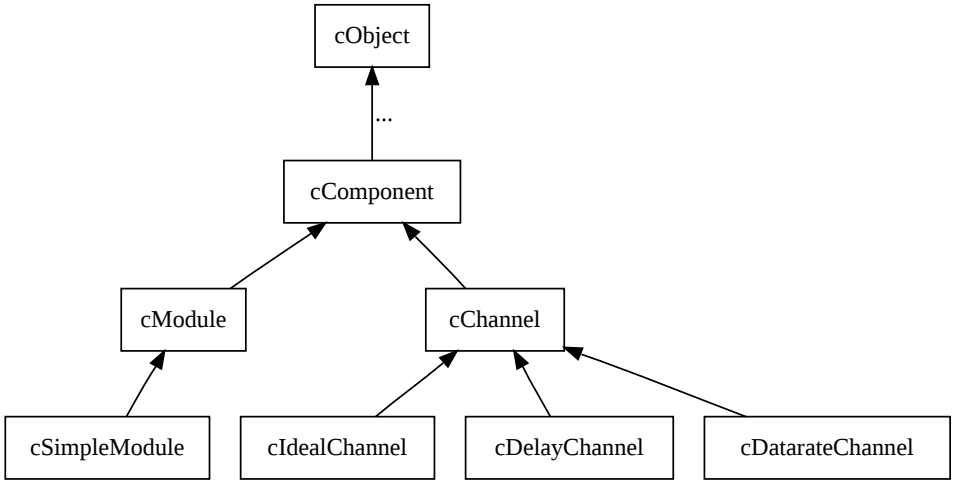


Figure 4.1: Inheritance of component, module and channel classes

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

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

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

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

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

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

- `handleMessage(cMessage *msg)`. It is invoked with the message as parameter whenever the module receives a message. `handleMessage()` is expected to *process* the message, and then return. Simulation time never elapses inside `handleMessage()` calls, only between them.
- `activity()` is started as a coroutine[1] at the beginning of the simulation, and it runs until the end of simulation (or until the function returns or otherwise terminates). Messages are obtained with `receive()` calls. Simulation time elapses inside `receive()` calls.

[1] Cooperatively scheduled thread, explained later.

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

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

4.3 Defining Simple Module Types

4.3.1 Overview

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

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

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

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

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

// register module class with OMNeT++
Define_Module(HelloModule);

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

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

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

```
// file: HelloModule.ned
simple HelloModule
{
```

```
    gates:
        input in;
}
```

4.3.2 Constructor

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

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

`cSimpleModule` itself has two constructors:

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

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

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

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

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

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

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

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

4.3.3 Initialization and Finalization

Basic Usage

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

The reason `initialize()` exists is that usually you cannot put simulation-related code into the simple module constructor, because the simulation model is still being setup when the

constructor runs, and many required objects are not yet available. In contrast, `initialize()` is called just before the simulation starts executing, when everything else has been set up already.

`finish()` is for recording statistics, and it only gets called when the simulation has terminated normally. It does not get called when the simulations stops with an error message. The destructor always gets called at the end, no matter how the simulation stopped, but at that time it is fair to assume that the simulation model has been halfway demolished already.

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

Constructor:

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

`initialize()`:

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

`finish()`:

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

Destructor:

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

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

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

Invocation Order

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

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

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

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

The calling order for `finish()` is the reverse of the order of `initialize()`: first submodules, then the encompassing compound module. [2]

This is summarized in the following pseudocode:

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

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

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

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

Multi-Stage Initialization

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

```
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

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

"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. [3]

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

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

4.4 Adding Functionality to cSimpleModule

This section discusses `cSimpleModule`'s four previously mentioned member functions, intended to be redefined by the user: `initialize()`, `handleMessage()`, `activity()` and `finish()`. A fifth, less frequently used method, `handleParameterChange`, is described in section 4.5.5.

4.4.1 handleMessage()

Function Called for Each Event

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

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

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

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

```
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()` and `wait()` functions in `handleMessage()`, because they are coroutine-based by nature, as explained in the section about `activity()`.

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

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

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

You can initialize these variables from the `initialize()` function. The constructor is not a very good place for this purpose, because it is called in the network setup phase when

the model is still under construction, so a lot of information you may want to use is not yet available.

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

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

Application Area

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

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

Example 1: Protocol Models

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

This usually results in the following source code pattern:

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

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

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

// ...

void FooProtocol::handleMessage(cMessage *msg)
```

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

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

Example 2: Simple Traffic Generators and Sinks

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

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

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

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

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

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

Define_Module(Generator);

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

void Generator::handleMessage(cMessage *msg)
```

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

Example 3: Bursty Traffic Generator

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

We add some data members to the class:

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

The code:

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

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

Define_Module(BurstyGenerator);

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

void BurstyGenerator::handleMessage(cMessage *msg)
{
    // generate & send packet
    cMessage *pkt = new cMessage;
    send(pkt, "out");
    // if this was the last packet of the burst
    if (--burstCounter == 0)
    {
        // schedule next burst
        burstCounter = burstLength;
```

```
            scheduleAt(simTime()+exponential(5.0), msg);
        }
        else
        {
            // schedule next sending within burst
            scheduleAt(simTime()+exponential(1.0), msg);
        }
    }
```

Pros and Cons of Using `handleMessage()`

Pros:

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

Cons:

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

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

Other Simulators

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

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

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

4.4.2 activity()

Process-Style Description

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

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

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

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

Application Area

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

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

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

        while (there is more to do)
        {
            send data on network connection
            if (connection broken)
            {
                continue outer loop // loop back to outer while()
            }
            wait(some time)
            receive data on network connection
            if (connection broken)
            {
                continue outer loop // loop back to outer while()
            }
            wait(some time)
        }
```

```
        close connection by sending CLOSE command to transport layer
        if (close not successful)
        {
            // handle error
        }
        wait(some time)
    }
}
```

If you have to handle several connections simultaneously, you may dynamically create them as instances of the simple module above. Dynamic module creation will be discussed later.

There are situations when you certainly *do not want* to use `activity()`. If your `activity()` function contains no `wait()` and it has only one `receive()` call at the top of an infinite loop, there is no point in using `activity()` and the code should be written with `handleMessage()`. The body of the infinite loop would then become the body to `handleMessage()`, state variables inside `activity()` would become data members in the module class, and you'd initialize them in `initialize()`.

Example:

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

should rather be programmed as:

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

Activity() Is Run as a Coroutine

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

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

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

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

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

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

Their pseudocode, as implemented in OMNeT++:

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

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

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

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

Starter Messages

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

Coroutine Stack Size

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

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

initialize() and finish() with activity()

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

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

Thus, a typical setup looks like this in pseudocode:

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

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

    while(true)
    {
        ...
    }
}

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

Pros and Cons of Using `activity()`

Pros:

- `initialize()` not needed, state can be stored in local variables of `activity()`

- process-style description is a natural programming model in some cases

Cons:

- limited scalability: coroutine stacks can unacceptably increase the memory requirements of the simulation program if you have several thousands or ten thousands of simple modules;
- run-time overhead: switching between coroutines is somewhat slower than a simple function call
- does not enforce a good programming style: using `activity()` tends to lead to unreliable, spaghetti code

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

Other Simulators

Coroutines are used by a number of other simulation packages:

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

4.4.3 How to Avoid Global Variables

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

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

4.4.4 Reusing Module Code via Subclassing

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

```
class TransportProtocolExt : public TransportProtocol
{
```

```
  protected:
    virtual void recalculateTimeout();
};

Define_Module(TransportProtocolExt);

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

The corresponding NED declaration:

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

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

4.5 Accessing Module Parameters

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

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

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

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

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

If you use the `par("foo")` parameter in expressions (such as `4*par("foo")+2`), the C++ compiler may be unable to decide between overloaded operators and report ambiguity. In that case you have to clarify by adding either an explicit cast (`(double)par("foo")` or `(long)par("foo")`) or use the `doubleValue()` or `longValue()` methods.

4.5.1 Volatile and Non-Volatile Parameters

A parameter can be declared `volatile` in the NED file. The `volatile` modifier indicates that a parameter is re-read every time a value is needed during simulation. Volatile parameters

typically are used for things like random packet generation interval, and are assigned values like `exponential(1.0)` (numbers drawn from the exponential distribution with mean 1.0).

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

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

```
class Source : public cSimpleModule
{
  protected:
    long numJobs;
    virtual void initialize();
    ...
};

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

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

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

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

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

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

void Source::handleMessage(cMessage *msg)
{
```

```
    ...
    scheduleAt(simTime() + intervalp->doubleValue(), timerMsg);
    ...
}
```

4.5.2 Changing a Parameter's Value

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

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

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

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

4.5.3 Further cPar Methods

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

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

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

An example usage of the above methods:

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

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

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

4.5.4 Emulating Parameter Arrays

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

The `cStringTokenizer` class can be quite useful for this purpose. The constructor accepts a string, which it regards as a sequence of tokens (words) separated by delimiter characters (by default, spaces). Then you can either enumerate the tokens and process them one by one (`hasMoreTokens()`, `nextToken()`), or use one of the `cStringTokenizer` convenience methods to convert them into a vector of strings (`asVector()`), integers (`asIntVector()`), or `double`s (`asDoubleVector()`).

The latter methods can be used like this:

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

and

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

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

The following example processes the string by enumerating the tokens:

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

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

4.5.5 handleParameterChange()

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

To enable notification, redefine the `handleParameterChange()` method of the module class. This method will be called back by the simulation kernel when a module parameter changes, *except during initialization of the given module*.

NOTE: Notifications are disabled during the initialization of the component, because they would make it very difficult to write components that work reliably under all conditions. `handleParameterChange()` is usually triggered from another module (it does not make much sense for a module to change its own parameters), so the relative order of `initialize()` and `handleParameterChange()` would be effectively determined by the initialization order of modules, which generally cannot be relied upon. After the last stage of the initialization of the component is finished, `handleParameterChange()` is called by the simulation kernel with `NULL` as a parameter name. This allows the component to react to parameter changes that occurred during the initialization phase.

The method signature is the following:

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

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

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

If your code heavily depends on notifications and you would like to receive notifications during initialization or finalization as well, one workaround is to explicitly call `handleParameterChange()` from the `initialize()` or `finish()` function:

```
for (int i=0; i<getNumParams(); i++)
    handleParameterChange(par(i).getName());
```

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

4.6 Accessing Gates and Connections

4.6.1 Gate Objects

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

Accessing Gates by Name

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

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

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

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

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

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

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

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

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

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

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

Gate Vectors

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

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

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

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

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

Gate IDs

A gate can also be accessed by its ID. A very important property of gate IDs is that they are *contiguous* within a gate vector, that is, the ID of a gate `g[k]` can be calculated as the ID of `g[0]` plus k. This allows you to efficiently access any gate in a gate vector, because retrieving a gate by ID is more efficient than by name and index. The index of the first gate can be obtained

with `gate("out",0)->getId()`, but it is better to use a dedicated method, `gateBaseId()`, because it also works when the gate vector size is zero.

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

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

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

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

Enumerating All Gates

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

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

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

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

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

Adding and Deleting Gates

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

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

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

cGate Methods

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

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

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

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

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

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

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

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

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

4.6.2 Connections

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

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

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

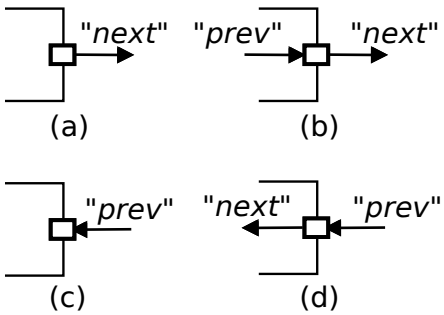


Figure 4.2: (a) simple module output gate, (b) compound module output gate, (c) simple module input gate, (d) compound module input gate

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

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

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

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

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

4.6.3 The Connection's Channel

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

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

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

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

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

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

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

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

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

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

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

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

Channels are covered in more detail in section 4.8.

4.7 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 – the rest of OMNeT++ exists 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. Network packets are represented with `cPacket`, which is also subclassed from `cMessage`. Message objects are created using the C++ `new` operator, and destroyed using the `delete` operator when they are no longer needed.

Messages are described in detail in chapter 5. At this point, all we need to know about them is that they are referred to as `cMessage *` pointers. In the examples below, messages will

be created with `new cMessage("foo")` where `"foo"` is a descriptive message name, used for visualization and debugging purposes.

4.7.1 Self-Messages

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

- A source module that periodically creates and sends messages needs to schedule the next send after every send operation;
- A server which processes jobs from a queue needs to start a timer every time it begins processing a job. When the timer expires, the finished job can be sent out, and a new job may start processing;
- When a packet is sent by a communications protocol that employs retransmission, it needs to schedule a timeout so that the packet can be retransmitted if no acknowledge arrives within a certain amount of time.

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

Scheduling an Event

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

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

Cancelling an Event

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

```
cancelEvent(msg);
```

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

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

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

```
void Protocol::handleMessage(cMessage *msg)
{
    if (msg == timeoutEvent) {
        // timeout expired, re-send packet and restart timer
        send(currentPacket->dup(), "out");
        scheduleAt(simTime() + timeout, timeoutEvent);
    }
    else if (...) {  // if acknowledgement received
        // cancel timeout, prepare to send next packet, etc.
        cancelEvent(timeoutEvent);
        ...
    }
    else {
       ...
    }
}
```

Re-scheduling an Event

If you want to reschedule an event which is currently scheduled to a different simulation time, first you have to cancel it using `cancelEvent()`. This is shown in the following example code:

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

4.7.2 Sending Messages

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

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

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

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

Examples:

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

To send via an inout gate, remember that an inout gate is an input and an output gate glued together, and the two halves can be identified with the `$i` and `$o` name suffixes. Thus, for sending you need to specify the gate name with the `$o` suffix:

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

4.7.3 Broadcasts and Retransmissions

When 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. Instead, duplicate the message object.

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

To enforce the rules above, all message sending functions check that you actually own the message you are about to send. If the message is in another module, currently scheduled or in a queue, etc., you'll get a runtime error: *not owner of message*. [4]

Broadcasting Messages

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

Example:

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

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

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

[4] 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.

Retransmissions

Many communication protocols involve retransmissions of packets (frames). When implementing retransmissions, you cannot just hold a pointer to the same message object and send it again and again – you'd get the *not owner of message* error on the first resend.

Instead, for (re)transmission, you should create and send copies of the message, and retain the original. When you are sure there will not be any more retransmission, you can delete the original message.

Creating and sending a copy:

```
// (re)transmit packet:
cMessage *copy = packet->dup();
send(copy, "out");
```

and finally (when no more retransmissions will occur):

```
delete packet;
```

4.7.4 Delayed Sending

Sometimes it is necessary for module to hold a message for some time interval, and then send it. This can be achieved with self-messages, but there is a more straightforward method: delayed sending. The following methods are provided for delayed sending:

```
sendDelayed(cMessage *msg, double delay, const char *gateName, int index);
sendDelayed(cMessage *msg, double delay, int gateId);
sendDelayed(cMessage *msg, double delay, cGate *gate);
```

The arguments are the same as for `send()`, except for the extra *delay* parameter. The delay value must be non-negative. The effect of the function is similar to as if the module had kept the message for the delay interval and sent it afterwards; even the *sending time* timestamp of the message will be set to the current simulation time plus *delay*.

A example call:

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

The `sendDelayed()` function does not internally perform a `scheduleAt()` followed by a `send()`, but rather it computes everything about the message sending up front, including the arrival time and the target module. This has two consequences. First, `sendDelayed()` is more efficient than a `scheduleAt()` followed by a `send()` because it eliminates one event. The second, less pleasant consequence is that changes in the connection path during the delay will *not* be taken into account (because everything is calculated in advance, before the changes take place).

> **NOTE:** The fact that `sendDelayed()` computes the message arrival information up front does not make a difference if the model is static, but may lead to surprising results if the model changes in time. For example, if a connection in the path gets deleted, disabled, or reconnected to another module during the delay period, the message will still be delivered to the original module as if nothing happened.
> Therefore, despite its performance advantage, you should think twice before using `sendDelayed()` in a simulation model. It may have its place in a one-shot simulation model that you know is static, but it certainly should be avoided in reusable modules that need to work correctly in a wide variety of simulation models.

4.7.5 Direct Message Sending

At times it is covenient to be able to send a message directly to an input gate of another module. The `sendDirect()` function is provided for this purpose.

This function has several flavors. The first set of `sendDirect()` functions accept a message and a target gate; the latter can be specified in various forms:

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

An example for direct sending:

```
cModule *targetModule = getParentModule()->getSubmodule("node2");
sendDirect(new cMessage("msg"), targetModule, "in");
```

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

The target gate must be an unconnected gate; in other words, modules must have dedicated gates to be able to receive messages sent via `sendDirect()`. You cannot have a gate which receives messages via both connections and `sendDirect()`.

It is recommended to tag gates dedicated for receiving messages via `sendDirect()` with the `@directIn` property in the module's NED declaration. This will cause OMNeT++ not to complain that the gate is not connected in the network or compound module where the module is used.

An example:

```
simple Radio {
    gates:
        input radioIn @directIn; // for receiving air frames
}
```

The target module is usually a simple module, but it can also be a compound module. The message will follow the connections that start at the target gate, and will be delivered to the module at the end of the path – just as with normal connections. The path must end in a simple module.

It is even permitted to send to an output gate, which will also cause the message to follow the connections starting at that gate. This can be useful, for example, when several submodules are sending to a single output gate of their parent module.

A second set of `sendDirect()` methods accept a propagation delay and a transmission duration as parameters as well:

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

The transmission duration parameter is important when the message is also a packet (instance of `cPacket`). For messages that are not packets (not subclassed from `cPacket`), the duration parameter is ignored.

If the message is a packet, the duration will be written into the packet, and can be read by the receiver with the `getDuration()` method of the packet.

The receiver module can choose whether it wants the simulation kernel to deliver the packet object to it at the start or at the end of the reception. The default is the latter; the module can change it by calling `setDeliverOnReceptionStart()` on the final input gate, that is, on `targetGate->getPathEndGate()`.

4.7.6 Packet Transmissions

When a message is sent out on a gate, it usually travels through a series of connections until it arrives at the destination module. We call this series of connections a *connection path*.

Several connections in the path may have an associated channel, but there can be only one channel per path that models nonzero transmission duration. This restriction is enforced by the simulation kernel. This channel is called the *transmission channel*. [5]

> **NOTE:** In practice, this means that there can be only one `ned.DatarateChannel` in the path. Note that unnamed channels with a `datarate` parameter also map to `ned.DatarateChannel`.

Transmitting a Packet

Packets may only be sent when the transmission channel is idle. This means that after each transmission, the sender module needs to wait until the channel has finished transmitting before it can send another packet.

You can get a pointer to the transmission channel by calling the `getTransmissionChannel()` method on the output gate. The channel's `isBusy()` and `getTransmissionFinishTime()` methods can tell you whether a channel is currently transmitting, and when the transmission is going to finish. (When the latter is less or equal the current simulation time, the channel is free.) If the channel is currently busy, sending needs to be postponed: the packet can be stored in a queue, and a timer (self-message) can be scheduled for the time when the channel becomes empty.

A code example to illustrate the above process:

```
cPacket *pkt = ...; // packet to be transmitted
cChannel *txChannel = gate("out")->getTransmissionChannel();
simtime_t txFinishTime = txChannel->getTransmissionFinishTime();
if (txFinishTime <= simTime())
{
    // channel free; send out packet immediately
    send(pkt, "out");
}
else
{
    // store packet and schedule timer; when the timer expires,
    // the packet should be removed from the queue and sent out
    txQueue.insert(pkt);
    scheduleAt(txFinishTime, endTxMsg);
```

[5] Moreover, if `sendDirect()` with a nonzero duration was used to send the packet to the start gate of the path, then the path cannot have a transmission channel at all. The point is that the a transission duration must be unambiguous.

```
}
```

NOTE: If there is a channel with a propagation delay in the path before the transmission channel, the delay should be manually substracted from the value returned by `getTransmissionFinishTime()`! The same applies to `isBusy()`: it tells whether the channel is currently busy, and not whether it will be busy when a packet that you send gets there. It is therefore advisable that you never use propagation delays in front of a transmission channel in a path.

The `getTransmissionChannel()` method searches the connection path each time it is called. If performance is important, it is recommended that you obtain the transmission channel pointer once, and cache it. When the network topology changes, the cached channel pointer needs to be updated; section 4.14.3 describes the mechanism that can be used to get notifications about topology changes.

Receiving a Packet

As a result of error modeling in the channel, the packet may arrive with the *bit error* flag set (`hasBitError() method`. It is the receiver module's responsibility to examine this flag and take appropriate action (i.e. discard the packet).

Normally the packet object gets delivered to the destination module at the simulation time that corresponds to finishing the reception of the message (ie. the arrival of its last bit). However, the receiver module may change this by "reprogramming" the receiver gate with the `setDeliverOnReceptionStart()` method:

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

This method may only be called on simple module input gates, and it instructs the simulation kernel to deliver packets arriving through that gate at the simulation time that corresponds to the beginning of the reception process. `getDeliverOnReceptionStart()` only needs to be called once, so it is usually done in the `initialize()` method of the module.

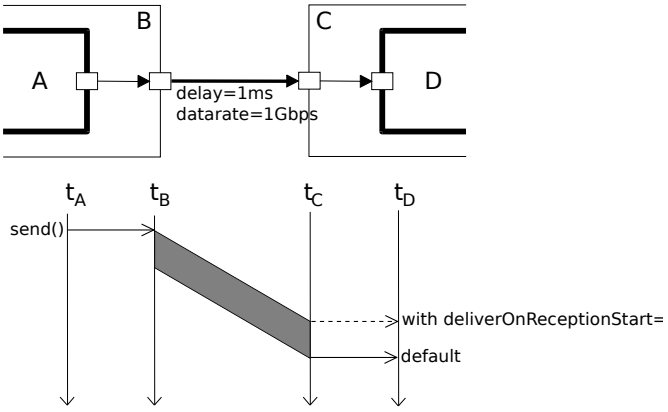


Figure 4.3: Packet transmission

When a packet is delivered to the module, the packet's `isReceptionStart()` method can be called to determine whether it corresponds to the start or end of the reception process

(it should be the same as the `getDeliverOnReceptionStart()` flag of the input gate), and `getDuration()` returns the transmission duration.

The following example code prints the start and end times of a packet reception:

```
simtime_t startTime, endTime;
if (pkt->isReceptionStart())
{
    // gate was reprogrammed with setDeliverOnReceptionStart(true)
    startTime = pkt->getArrivalTime(); // or: simTime();
    endTime = startTime + pkt->getDuration();
}
else
{
    // default case
    endTime = pkt->getArrivalTime(); // or: simTime();
    startTime = endTime - pkt->getDuration();
}
ev << "interval: " << startTime << ".." << endTime << "\n";
```

Note that this works with wireless connections (`sendDirect()`) as well; there, the duration is an argument to the `sendDirect()` call.

Aborting Transmissions

Sometimes you want the sender to abort transmission. The support OMNeT++ provides for this task is the `forceTransmissionFinishTime()` method of channels. This method forcibly overwrites the *transmissionFinishTime* member of the channel with the given value, allowing the sender to transmit another packet without raising the *"channel is currently busy"* error. The receiving party needs to be notified about the aborted transmission by some user-defined means, for example by sending another packet or an out-of-band message.

Implementation of Message Sending

Message sending is implemented like this: the arrival time and the bit error flag of a message are calculated right inside the `send()` call, then the message is inserted into the FES with the calculated arrival time. The message does *not* get scheduled individually for each link. This implementation was chosen because of its run-time efficiency.

> **NOTE:** The consequence of this implementation is that any change in the channel's parameters (delay, data rate, bit error rate, etc.) will only affect messages *sent* after the change. Messages already underway will not be influenced by the change.
> This is not a huge problem in practice, but if it is important to model channels with changing parameters, the solution is to insert simple modules into the path to ensure strict scheduling.

4.7.7 Receiving Messages with activity()

Receiving Messages

`activity()`-based modules receive messages with the `receive()` method of `cSimpleModule`. `receive()` cannot be used with `handleMessage()`-based modules.

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

The `receive()` function accepts an optional *timeout* parameter. (This is a *delta*, not an absolute simulation time.) If no message arrives within the timeout period, the function returns a `NULL` pointer. [6]

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

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

The wait() Function

The `wait()` function suspends the execution of the module for a given amount of simulation time (a *delta*). `wait()` cannot be used with `handleMessage()`-based modules.

```
wait(delay);
```

In other simulation software, `wait()` is often called *hold*. Internally, the `wait()` function is implemented by a `scheduleAt()` followed by a `receive()`. The `wait()` function is very convenient in modules that do not need to be prepared for arriving messages, for example message generators. An example:

```
for (;;)
{
    // wait for some, potentially random, amount of time, specified
    // in the interarrivalTime volatile module parameter
    wait(par("interarrivalTime").doubleValue());

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

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

```
cQueue queue("queue");
...
waitAndEnqueue(waitTime, &queue);
if (!queue.empty())
{
```

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

```
    // process messages arrived during wait interval
    ...
}
```

4.8 Channels

4.8.1 Overview

Channels encapsulate parameters and behavior associated with connections. Channel types are like simple modules, in the sense that they are declared in NED, and there are C++ implementation classes behind them. Section 3.5 describes NED language support for channels, and explains how to associate C++ classes with channel types declared in NED.

C++ channel classes must subclass from the abstract base class `cChannel`. However, when creating a new channel class, it may be more practical to extend one of the existing C++ channel classes behind the three predefined NED channel types:

- `cIdealChannel` implements the functionality of `ned.IdealChannel`
- `cDelayChannel` implements the functionality of `ned.DelayChannel`
- `cDatarateChannel` implements the functionality of `ned.DatarateChannel`

Channel classes need to be registered with the `Define_Channel()` macro, just like simple module classes need `Define_Module()`.

The channel base class `cChannel` inherits from `cComponent`, so channels participate in the initialization and finalization protocol (`initialize()` and `finish()`) described in 4.3.3.

The parent module of a channel (as returned by the `getParentModule()`) is the module that contains the connection. If a connection connects two modules that are children of the same compound module, the channel's parent is the compound module. If the connection connects a compound module to one of its submodules, the channel's parent is also the compound module.

4.8.2 The Channel API

When subclassing `Channel`, you have to redefine and provide implementations for the following pure virtual member functions:

- `bool isTransmissionChannel() const`
- `simtime_t getTransmissionFinishTime() const`
- `void processMessage(cMessage *msg, simtime_t t, result_t& result)`

The first two functions are usually one-liners; the channel behavior is encapsulated in the third function, `processMessage()`.

Transmission Channels

The first function, `isTransmissionChannel()`, determines whether the channel is a *transmission channel*, i.e. one that models transmission duration. A transmission channel sets the duration field of packets sent through it (see the `setDuration()` field of `cPacket`).

The `getTransmissionFinishTime()` function is only used with transmission channels, and it should return the simulation time the sender will finish (or has finished) transmitting. This method is called by modules that send on a transmission channel to find out when the channel becomes available. The channel's `isBusy()` method is implemented simply as `return getTransmissionFinishTime() < simTime()`. For non-transmission channels, the `getTransmissionFinishTime()` return value may be any simulation time which is less than or equal to the current simulation time.

The processMessage() Function

The third function, `processMessage()` encapsulates the channel's functionality. However, before going into the details of this function we need to understand how OMNeT++ handles message sending on connections.

Inside the `send()` call, OMNeT++ follows the connection path denoted by the `getNextGate()` functions of gates, until it reaches the target module. At each "hop", the corresponding connection's channel (if the connection has one) gets a chance to add to the message's arrival time (*propagation time modeling*), calculate a *transmission duration*, and to modify the message object in various ways, such as set the bit error flag in it (*bit error modeling*). After processing all hops that way, OMNeT++ inserts the message object into the Future Events Set (FES, see section 4.1.2), and the `send()` call returns. Then OMNeT++ continues to process events in increasing timestamp order. The message will be delivered to the target module's `handleMessage()` (or `receive()`) function when it gets to the front of the FES.

A few more details: a channel may instruct OMNeT++ to delete the message instead of inserting it into the FES; this can be useful to model disabled channels, or to model that the message has been lost altogether. The `getDeliverOnReceptionStart()` flag of the final gate in the path will determine whether the transmission duration will be added to the arrival time or not. Packet transmissions have been described in section 4.7.6.

Now, back to the `processMessage()` method.

The method gets called as part of the above process, when the message is processed at the given hop. The method's arguments are the message object, the simulation time the beginning of the message will reach the channel (i.e. the sum of all previous propagation delays), and a struct in which the method can return the results.

The `result_t` struct is an inner type of `cChannel`, and looks like this:

```
struct result_t {
    simtime_t delay;      // propagation delay
    simtime_t duration;   // transmission duration
    bool discard;         // whether the channel has lost the message
};
```

It also has a constructor that initializes all fields to zero; it is left out for brevity.

The method should model the transmission of the given message starting at the given *t* time, and store the results (propagation delay, transmission duration, deletion flag) in the result object. Only the relevant fields in the result object need to be changed, others can be left untouched.

Transmission duration and bit error modeling only applies to packets (i.e. to instances of `cPacket`, where `cMessage`'s `isPacket()` returns true); it should be skipped for non-packet messages. `processMessage()` does not need to call the `setDuration()` method on the packet; this is done by the simulation kernel. However, it should call `setBitError(true)` on the packet if error modeling results in bit errors.

If the method sets the `discard` flag in the result object, that means that the message object will be deleted by OMNeT++; this facility can be used to model that the message gets lost in the channel.

The `processMessage()` method does not need to throw error on overlapping transmissions, or if the packet's duration field is already set; these checks are done by the simulation kernel before `processMessage()` is called.

4.8.3 Channel Examples

To illustrate coding channel behavior, we look at how the built-in channel types are implemented.

`cIdealChannel` lets through messages and packets without any delay or change. Its `isTransmissionChannel()` method returns `false`, `getTransmissionFinishTime()` returns $0s$, and the body of its `processMessage()` method is empty:

```
void cIdealChannel::processMessage(cMessage *msg, simtime_t t, result_t& result)
{
}
```

`cDelayChannel` implements propagation delay, and it can be disabled; in its disabled state, messages sent though it will be discarded. This class still models zero transmission duration, so its `isTransmissionChannel()` and `getTransmissionFinishTime()` methods still return `false` and $0s$. The `processMessage()` method sets the appropriate fields in the `result_t` struct:

```
void cDelayChannel::processMessage(cMessage *msg, simtime_t t, result_t& result)
{
    // if channel is disabled, signal that message should be deleted
    result.discard = isDisabled;

    // propagation delay modeling
    result.delay = delay;
}
```

The `handleParameterChange()` method is also redefined, so that the channel can update its internal `delay` and `isDisabled` data members if the corresponding channel parameters change during simulation. [7]

`cDatarateChannel` is different. It performs model packet duration (duration is calculated from the data rate and the length of the packet), so `isTransmissionChannel()` returns `true`. `getTransmissionFinishTime()` returns the value of a `txfinishtime` data member, which gets updated after every packet.

```
simtime_t cDatarateChannel::getTransmissionFinishTime() const
{
    return txfinishtime;
}
```

[7] This code is a little simplified; the actual code uses a bit in a bitfield to store the value of `isDisabled`.

cDatarateChannel's processMessage() method makes use of the isDisabled, datarate, ber and per data members, which are also kept up to date with the help of handleParameterChange().

```
void cDatarateChannel::processMessage(cMessage *msg, simtime_t t, result_t& result)
{
    // if channel is disabled, signal that message should be deleted
    if (isDisabled) {
        result.discard = true;
        return;
    }

    // datarate modeling
    if (datarate!=0 && msg->isPacket()) {
        simtime_t duration = ((cPacket *)msg)->getBitLength() / datarate;
        result.duration = duration;
        txfinishtime = t + duration;
    }
    else {
        txfinishtime = t;
    }

    // propagation delay modeling
    result.delay = delay;

    // bit error modeling
    if ((ber!=0 || per!=0) && msg->isPacket())
    {
        cPacket *pkt = (cPacket *)msg;
        if (ber!=0 && dblrand() < 1.0 - pow(1.0-ber, (double)pkt->getBitLength())
            pkt->setBitError(true);
        if (per!=0 && dblrand() < per)
            pkt->setBitError(true);
    }
}
```

4.9 Stopping the Simulation

4.9.1 Normal Termination

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

```
endSimulation();
```

endSimulation() is rarely needed in practice because you can specify simulation time and CPU time limits in the ini file (see later).

4.9.2 Raising Errors

If your simulation encounters an error condition, you can throw a cRuntimeError exception to terminate the simulation with an error message (and in case of Cmdenv, a nonzero

exit code). The `cRuntimeError` class has a constructor whose argument list is similar to `printf()`:

```
if (windowSize <= 0)
    throw cRuntimeError("Invalid window size %d; must be >=1", windowSize);
```

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

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

```
if (windowSize <= 0)
    error("Invalid window size %d; must be >=1", windowSize);
```

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

4.10 Finite State Machines

Overview

Finite State Machines (FSMs) can make life with `handleMessage()` easier. OMNeT++ provides a class and a set of macros to build FSMs.

The key points are:

- There are two kinds of states: *transient* and *steady*. On each event (that is, at each call to `handleMessage()`), the FSM transitions out of the current (*steady*) state, undergoes a series of state changes (runs through a number of *transient* states), and finally arrives at another *steady* state. Thus between two events, the system is always in one of the steady states. Transient states are therefore not really a must – they exist only to group actions to be taken during a transition in a convenient way.
- You can assign program code to handle entering and leaving a state (known as entry/exit code). Staying in the same state is handled as leaving and re-entering the state.
- Entry code should not modify the state (this is verified by 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 becomes too large.

The FSM API

FSM state is stored in an object of type `cFSM`. The possible states are defined by an enum; the enum is also a place to define which state is transient and which is steady. In the following example, SLEEP and ACTIVE are steady states and SEND is transient (the numbers in parentheses must be unique within the state type and they are used for constructing the numeric IDs for the states):

```
enum {
  INIT = 0,
  SLEEP = FSM_Steady(1),
  ACTIVE = FSM_Steady(2),
```

```
  SEND = FSM_Transient(1),
};
```

The actual FSM is embedded in a switch-like statement, `FSM_Switch()`, 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).getName()
      << ((exiting) ? ": leaving state " : ": entering state ")
      << (fsm).getStateName() << endl)
```

Implementation

The `FSM_Switch()` is a macro. It expands to a `switch()` statement embedded in a `for()` loop which repeats until the FSM reaches a steady state. (The actual code is rather scary, but if you are dying to see it, it is in `cfsm.h`.)

Infinite loops are avoided by counting state transitions: if an FSM goes through 64 transitions without reaching a steady state, the simulation will terminate with an error message.

An Example

Let us write another bursty packet generator. It will have two states, SLEEP and ACTIVE. In the SLEEP state, the module does nothing. In the ACTIVE state, it sends messages with a given inter-arrival time. The code was taken from the Fifo2 sample simulation.

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

class BurstyGenerator : public cSimpleModule
{
  protected:
    // parameters
    double sleepTimeMean;
    double burstTimeMean;
    double sendIATime;
    cPar *msgLength;

    // FSM and its states
    cFSM fsm;
    enum {
      INIT = 0,
      SLEEP = FSM_Steady(1),
      ACTIVE = FSM_Steady(2),
      SEND = FSM_Transient(1),
    };

    // variables used
    int i;
    cMessage *startStopBurst;
    cMessage *sendMessage;

    // the virtual functions
    virtual void initialize();
```

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

Define_Module(BurstyGenerator);

void BurstyGenerator::initialize()
{
    fsm.setName("fsm");
    sleepTimeMean = par("sleepTimeMean");
    burstTimeMean = par("burstTimeMean");
    sendIATime = par("sendIATime");
    msgLength = &par("msgLength");
    i = 0;
    WATCH(i); // always put watches in initialize()
    startStopBurst = new cMessage("startStopBurst");
    sendMessage = new cMessage("sendMessage");
    scheduleAt(0.0,startStopBurst);
}

void BurstyGenerator::handleMessage(cMessage *msg)
{
   FSM_Switch(fsm)
   {
     case FSM_Exit(INIT):
       // transition to SLEEP state
       FSM_Goto(fsm,SLEEP);
       break;
     case FSM_Enter(SLEEP):
       // schedule end of sleep period (start of next burst)
       scheduleAt(simTime()+exponential(sleepTimeMean),
                  startStopBurst);
     break;
     case FSM_Exit(SLEEP):
       // schedule end of this burst
       scheduleAt(simTime()+exponential(burstTimeMean),
                  startStopBurst);
       // transition to ACTIVE state:
       if (msg!=startStopBurst) {
         error("invalid event in state ACTIVE");
       }
       FSM_Goto(fsm,ACTIVE);
       break;
     case FSM_Enter(ACTIVE):
       // schedule next sending
       scheduleAt(simTime()+exponential(sendIATime), sendMessage);
     break;
     case FSM_Exit(ACTIVE):
       // transition to either SEND or SLEEP
       if (msg==sendMessage) {
         FSM_Goto(fsm,SEND);
       } else if (msg==startStopBurst) {
         cancelEvent(sendMessage);
```

```
          FSM_Goto(fsm,SLEEP);
        } else {
          error("invalid event in state ACTIVE");
        }
        break;
      case FSM_Exit(SEND):
      {
        // generate and send out job
        char msgname[32];
        sprintf( msgname, "job-%d", ++i);
        ev << "Generating " << msgname << endl;
        cMessage *job = new cMessage(msgname);
        job->setBitLength( (long) *msgLength );
        job->setTimestamp();
        send( job, "out" );
        // return to ACTIVE
        FSM_Goto(fsm,ACTIVE);
        break;
      }
    }
}
```

4.11 Navigating the Module Hierarchy

Module Vectors

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

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

Module IDs

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

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

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

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

Module IDs are guaranteed to be unique for the duration of the whole simulation, even when modules are created and destroyed dynamically; that is, IDs of deleted modules are not reused for newly created modules.

Walking Up and Down the Module Hierarchy

The surrounding compound module can be accessed by the `getParentModule()` member function:

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

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

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

`cModule`'s `findSubmodule()` and `getSubmodule()` member functions make it possible to look up the module's submodules by name (or name+index if the submodule is in a module vector). The first one returns the numeric module ID of the submodule, and the latter returns the module pointer. If the submodule is not found, they return -1 or NULL, respectively.

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

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

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

would give the same result as

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

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

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

Iterating over Submodules

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

For example:

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

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

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

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

Walking Along Links

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

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

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

4.12 Direct Method Calls Between Modules

In some simulation models, there might be modules which are too tightly coupled for message-based communication to be efficient. In such cases, the solution might be calling one simple module's public C++ methods from another module.

Simple modules are C++ classes, so normal C++ method calls will work. Two issues need to be mentioned, however:

- how to get a pointer to the object representing the module;
- how to let the simulation kernel know that a method call across modules is taking place.

Typically, the called module is in the same compound module as the caller, so the `getParentModule()` and `getSubmodule()` methods of `cModule` can be used to get a `cModule*` pointer to the called module. (Further ways to obtain the pointer are described in the section 4.11.) The `cModule*` pointer then has to be cast to the actual C++ class of the module, so that its methods become visible.

This makes the following code:

```
cModule *calleeModule = getParentModule()->getSubmodule("callee");
Callee *callee = check_and_cast<Callee *>(calleeModule);
callee->doSomething();
```

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

The second issue is how to let the simulation kernel know that a method call across modules is taking place. Why is this necessary in the first place? First, the simulation kernel always has to know which module's code is currently executing, in order for ownership handling and other internal mechanisms to work correctly. Second, the Tkenv simulation GUI can animate method calls, but to be able to do that, it has to know about them. Third, method calls are also recorded in the event log.

The solution is to add the `Enter_Method()` or `Enter_Method_Silent()` macro at the top of the methods that may be invoked from other modules. These calls perform context switching, and, in case of `Enter_Method()`, notify the simulation GUI so that animation of the method call can take place. `Enter_Method_Silent()` does not animate the method call, but otherwise it is equivalent `Enter_Method()`. Both macros accept a `printf()`-like argument list (it is optional for `Enter_Method_Silent()`), which should produce a string with the method name and the actual arguments as much as practical. The string is displayed in the animation (`Enter_Method()` only) and recorded into the event log.

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

4.13 Dynamic Module Creation

4.13.1 When Do You Need Dynamic Module Creation

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

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

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

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

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

4.13.2 Overview

To understand how dynamic module creation works, you have to know a bit about how OMNeT++ normally instantiates modules. Each module type (class) has a corresponding factory object of the class `cModuleType`. This object is created under the hood by the `Define_Module()` macro, and it has a factory method which can instantiate the module class (this function basically only consists of a `return new <moduleclass>(...)` statement).

The `cModuleType` object can be looked up by its name string (which is the same as the module class name). Once you have its pointer, it is possible to call its factory method and create an instance of the corresponding module class – without having to include the C++ header file containing module's class declaration into your source file.

The `cModuleType` object also knows what gates and parameters the given module type has to have. (This info comes from NED files.)

Simple modules can be created in one step. For a compound module, the situation is more complicated, because its internal structure (submodules, connections) may depend on parameter values and gate vector sizes. Thus, for compound modules it is generally required to first create the module itself, second, set parameter values and gate vector sizes, and then call the method that creates its submodules and internal connections.

As you know already, simple modules with `activity()` need a starter message. For statically created modules, this message is created automatically by OMNeT++, but for dynamically created modules, you have to do this explicitly by calling the appropriate functions.

Calling `initialize()` has to take place after insertion of the starter messages, because the

initializing code may insert new messages into the FES, and these messages should be processed *after* the starter message.

4.13.3 Creating Modules

The first step is to find the factory object. The `cModuleType::get()` function expects a fully qualified NED type name, and returns the factory object:

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

The return value does not need to be checked for `NULL`, because the function raises an error if the requested NED type was not found. (If this behavior is not what you need, you can use the similar `cModuleType::find()` function, which returns `NULL` if the type was not found.)

The All-in-One Method

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

```
cModule *mod = moduleType->createScheduleInit("node", this);
```

`createScheduleInit()` performs the following steps: `create()`, `finalizeParameters()`, `buildInside()`, `scheduleStart(now)` and `callInitialize()`.

This method can be used for both simple and compound modules. Its applicability is somewhat limited, however: because it does everything in one step, you do not have the chance to set parameters or gate sizes, and to connect gates before `initialize()` is called. (`initialize()` expects all parameters and gates to be in place and the network fully built when it is called.) Because of the above limitation, this function is mainly useful for creating basic simple modules.

The Detailed Procedure

If the `createScheduleInit()` all-in-one method is not applicable, one needs to use the full procedure. It consists of five steps:

1. Find the factory object;
2. Create the module;
3. Set up its parameters and gate sizes as needed;
4. Tell the (possibly compound) module to recursively create its internal submodules and connections;
5. Schedule activation message(s) for the new simple module(s).

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

See the following example, where Step 3 is omitted:

```
// find factory object
cModuleType *moduleType = cModuleType::get("foo.nodes.WirelessNode");
```

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

// create activation message
module->scheduleStart(simTime());
```

If you want to set up parameter values or gate vector sizes (Step 3.), the code goes between the `create()` and `buildInside()` calls:

```
// create
cModuleType *moduleType = cModuleType::get("foo.nodes.WirelessNode");
cModule *module = moduleType->create("node", this);

// set up parameters and gate sizes before we set up its submodules
module->par("address") = ++lastAddress;
module->finalizeParameters();

module->setGateSize("in", 3);
module->setGateSize("out", 3);

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

4.13.4 Deleting Modules

To delete a module dynamically, use `cModule`'s `deleteModule()` member function:

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

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

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

4.13.5 Module Deletion and finish()

`finish()` is called for *all* modules at the end of the simulation, no matter how the modules were created. If a module is dynamically deleted before that, `finish()` will not be invoked (`deleteModule()` does not do it). However, you can still manually invoke it before `deleteModule()`.

You can use the `callFinish()` function to invoke `finish()` (It is not a good idea to invoke `finish()` directly). If you are deleting a compound module, `callFinish()` will recursively invoke `finish()` for all submodules, and if you are deleting a simple module from another module, `callFinish()` will do the context switch for the duration of the call. [8]

Example:

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

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

4.13.6 Creating Connections

Connections can be created using `cGate`'s `connectTo()` method. [9] `connectTo()` should be invoked on the source gate of the connection, and expects the destination gate pointer as an argument:

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

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

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

```
cModuleType *moduleType = cModuleType::get("TicToc");
cModule *a = modtype->createScheduleInit("a", this);
cModule *b = modtype->createScheduleInit("b", this);

a->gate("out")->connectTo(b->gate("in"));
b->gate("out")->connectTo(a->gate("in"));
```

`connectTo()` also accepts a channel object (`cChannel*`) as an additional, optional argument. Similarly to modules, channels can be created using their factory object of the type `cChannelType`:

```
cGate *outg=..., *ing=...;

// find factory object and create a channel
cChannelType *channelType = cChannelType::get("foo.util.Channel");
cChannel *channel = channelType->create("channel");

// create connecting
outg->connectTo(ing, channel);
```

The channel object will be owned by the source gate of the connection, and you cannot reuse the same channel object with several connections.

If you need one of the built-in channel types (`cIdealChannel`, `cDelayChannel` or `cDatarateChannel`), the step to find the factory object can be spared, as those classes have static `create()` functions to create a channel instance.

`cDatarateChannel` also has member functions to set up its parameters: `setDelay()`, `setBitErrorRate()`, `setPacketErrorRate()` and `setDatarate()`.

An example that sets up a channel with a delay:

```
cDatarateChannel *channel = cDatarateChannel::create("channel");
channel->setDelay(0.001);

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

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

4.13.7 Removing Connections

The `disconnect()` method of `cGate` can be used to remove connections. This method has to be invoked on the *source* side of the connection. It also destroys the channel object associated with the connection, if one has been set.

```
srcGate->disconnect();
```

4.14 Signals

This section describes *simulation signals*, or signals for short. Signals are a versatile concept that first appeared in OMNeT++ 4.1.

Simulation signals can be used for:

- exposing statistical properties of the model, without specifying whether and how to record them
- receiving notifications about simulation model changes at runtime, and acting upon them
- implementing a publish-subscribe style communication among modules; this is advantageous when the producer and consumer of the information do not know about each other, and possibly there is many-to-one or many-to-many relationship among them
- emitting information for other purposes, for example as input for custom animation effects

Signals are emitted by components (modules and channels). Signals propagate on the module hierarchy up to the root. At any level, one can register listeners (callback objects); these listeners will be notified (called back) whenever a signal value is emitted. The result of upwards propagation is that listeners registered at a compound module can receive signals from all components in that submodule tree. A listener registered at the system module can receive signals from the whole simulation.

> **NOTE:** A channel's parent is the (compound) module that contains the connection, not the owner of either gate the channel is connected to.

Signals are identified by signal names (i.e. strings), but for efficiency reasons at runtime we use dynamically assigned numeric identifiers (signal IDs, typedef'd as `simsignal_t`). The mapping of signal names to signal IDs is global, so all modules and channels asking to resolve a particular signal name will get back the same numeric signal ID.

Listeners can subscribe to signal names or IDs, regardless of their source. For example, if two different and unrelated module types, say `Queue` and `Buffer`, both emit a signal named `"length"`, then a listener that subscribes to `"length"` at some higher compound module will get notifications from both `Queue` and `Buffer` module instances. The listener can still look at the source of the signal if it wants to distinguish the two (it is available as a parameter to the callback function), but the signals framework itself does not have such a feature.

NOTE: Because the component type that emits the signal is not part of the signal's identity, it is advised to choose signal names carefully. A good naming scheme facilitates "merging" of signals that arrive from different sources but mean the same thing, and reduces the chance of collisions between signals that accidentally have the same name but represent different things.

When a signal is emitted, it can carry a value with it. This is realized via overloaded `emit()` methods in components, and overloaded `receiveSignal()` methods in listeners. The signal value can be of selected primitive types, or an object pointer; anything that is not feasible to emit as a primitive type may be wrapped into an object, and emitted as such.

4.14.1 Design Considerations and Rationale

The implementation of signals is based on the following assumptions:

- subscribe/unsubscribe operations are rare compared to `emit()` calls, so it is `emit()` that needs to be efficient
- the signals mechanism is present in every module, so per-module memory overhead must be kept as low as possible
- it is expected that modules and channels will be heavily instrumented with signals, and only a subset of signals will actually be used (will have listeners) in any particular simulation; therefore, the CPU and memory overhead of momentarily unused signals must be as low as possible

These goals have been achieved in the 4.1 version with the following implementation. First, the data structure that used to store listeners in components is dynamically allocated, so if there are no listeners, the per-component overhead is only the size of the pointer (which will be `NULL` then).

Second, additionally there are two bitfields in every component that store which one of the first 64 signals (IDs 0..63) have local listeners and listeners in ancestor modules.[10] Using these bitfields, it is possible to determine in constant time for the first 64 signals whether the signal has listeners, so `emit()` can return immediately if there are none. For other signals, `emit()` needs to examine the listener lists up to the root every time. Even if a simulation uses more than 64 signals, in performance-critical situations it is possible to arrange that frequently emitted signals (e.g. `"txBegin"`) get the "fast" signal IDs, while infrequent signals (like e.g. `"routerDown"`) get the rest.

4.14.2 The Signals Mechanism

Signal-related methods are declared on `cComponent`, so they are available for both `cModule` and `cChannel`.

Signal IDs

Signals are identified by names, but internally numeric signal IDs are used for efficiency. The `registerSignal()` method takes a signal name as parameter, and returns the corresponding

[10] It is assumed that there will be typically less than 64 frequently used signals used at a time in a simulation.

simsignal_t value. The method is static, illustrating the fact that signal names are global. An example:

```
simsignal_t lengthSignalId = registerSignal("length");
```

The getSignalName() method (also static) does the reverse: it accepts a simsignal_t, and returns the name of the signal as const char * (or NULL for invalid signal handles):

```
const char *signalName = getSignalName(lengthSignalId); // --> "length"
```

NOTE: Since OMNeT++ 4.3, the lifetime of signal IDs is the entire program, and it is possible to call registerSignal() from initializers of global variables, e.g. static class members. In earlier versions, signal IDs were usually allocated in initialize(), and were only valid for that simulation run.

Emitting Signals

The emit() family of functions emit a signal from the module or channel. They take two parameters, the signal ID (simsignal_t) and the value:

```
emit(lengthSignalId, queue.length());
```

The value can be of type bool, long, double, simtime_t, const char *, or (const) cObject *. Other types can be cast into one of these types, or wrapped into an object subclassed from cObject.

When there are no listeners, the runtime cost of emit() is usually minimal. However, if producing a value has a significant runtime cost, then the mayHaveListeners() or hasListeners() method can be used to check beforehand whether the given signal has any listeners at all – if not, emitting the signal can be skipped. For some signals (in OMNeT++ 4.3, the first 64 signals used), the information whether it has listeners is cached per component, and can be produced in constant time.

Example usage:

```
if (mayHaveListeners(distanceToTargetSignal))
{
    double d = sqrt((x-targetX)*(x-targetX) + (y-targetY)*(y-targetY));
    emit(distanceToTargetSignal, d);
}
```

The mayHaveListeners() method is very efficient (a constant-time operation) because it only uses this cached information; if the state is not cached for the signal, it just returns true. In contrast, hasListeners() will search up to the top of the module tree if the answer is not cached, so it is generally slower. We recommend that you take into account the cost of producing notification information when deciding between mayHaveListeners() and hasListeners().

Signal Declarations

Since OMNeT++ 4.4, signals can be declared in NED files for documentation purposes, and OMNeT++ can check that only declared signals are emitted, and that they actually conform to the declarations (with regard to the data type, etc.)

The following example declares a queue module that emits a signal named queueLength:

```
simple Queue
{
    parameters:
        @signal[queueLength](type=long);
        ...
}
```

As you can see, signals are declared with the `@signal` property on the module or channel that emits it. (NED properties are described in 3.12). The property index corresponds to the signal name, and the property's body may declare various attributes of the signal; currently only the data type is supported.

The `type` property key is optional; when present, its value should be `bool`, `long`, `unsigned long`, `double`, `simtime_t`, `string`, or a registered class name optionally followed by a question mark. Classes can be registered using the `Register_Class()` or `Register_Abstract_Class()` macros; these macros create a `cObjectFactory` instance, and the simulation kernel will call `cObjectFactory`'s `isInstance()` method to check that the emitted object is really a subclass of the declared class. `isInstance()` just wraps a C++ `dynamic_cast`.)

A question mark after the class name means that the signal is allowed to emit `NULL` pointers. For example, a module named `PPP` may emit the frame (packet) object every time it starts transmiting, and emit `NULL` when the transmission is completed:

```
simple PPP
{
    parameters:
        @signal[txFrame](type=PPPFrame?);  // a PPPFrame or NULL
        ...
}
```

The property index may contain wildcards, which is important if you want to declare signals whose names are only known at runtime. For example, if a module emits signals called `session-1-seqno`, `session-2-seqno`, `session-3-seqno`, etc. for the individual sessions it handles, you can declare those signals as:

```
    @signal[session-*-seqno]();
```

Enabling Signal Checking

In OMNeT++ 4.x, signal checking is turned off by default. You can turn it on with the **check-signals** configuration option in `omnetpp.ini`:

```
check-signals = true
```

It is expected that starting with OMNeT++ 5.0, signal checking will be turned on by default when the simulation kernel is compiled in debug mode. It will continue to be turned off in release mode simulation kernels due to performance reasons.

Signal Data Objects

When emitting a signal with a `cObject*` pointer, you can pass as data an object that you already have in the model, provided you have a suitable object at hand. However, it is often necessary to declare a custom class to hold all the details, and fill in an instance just for the purpose of emitting the signal.

The custom notification class must be derived from `cObject`. We recommend that you also add `noncopyable` as a base class, because then you don't need to write a copy constructor, assignment operator, and `dup()` function, sparing some work. When emitting the signal, you can create a temporary object, and pass its pointer to the `emit()` function.

An example of custom notification classes is the firing of model change notifications (see 4.14.3). The data class that accompanies a signal that announces that a gate or gate vector is about to be created looks like this:

```
class cPreGateAddNotification : public cObject, noncopyable
{
  public:
    cModule *module;
    const char *gateName;
    cGate::Type gateType;
    bool isVector;
};
```

And the code that emits the signal:

```
if (hasListeners(PRE_MODEL_CHANGE))
{
    cPreGateAddNotification tmp;
    tmp.module = this;
    tmp.gateName = gatename;
    tmp.gateType = type;
    tmp.isVector = isVector;
    emit(PRE_MODEL_CHANGE, &tmp);
}
```

Subscribing to Signals

The `subscribe()` method registers a listener for a signal. Listeners are objects that extend the `cIListener` class. The same listener object can be subscribed to multiple signals. `subscribe()` has two arguments: the signal and a pointer to the listener object:

```
cIListener *listener = ...;
simsignal_t lengthSignalId = registerSignal("length");
subscribe(lengthSignalId, listener);
```

For convenience, the `subscribe()` method has a variant that takes the signal name directly, so the `registerSignal()` call can be omitted:

```
cIListener *listener = ...;
subscribe("length", listener);
```

One can also subscribe at other modules, not only the local one. For example, in order to get signals from all parts of the model, one can subscribe at the system module level:

```
cIListener *listener = ...;
simulation.getSystemModule()->subscribe("length", listener);
```

The `unsubscribe()` method has the same parameter list as `subscribe()`, and unregisters the given listener from the signal:

```
unsubscribe(lengthSignalId, listener);
```

or

```
unsubscribe("length", listener);
```

It is an error to subscribe the same listener to the same signal twice.

> **NOTE:** When a listener is deleted, it must already be unsubscribed from all components it has subscribed to. This is explained in 4.14.2.

It is possible to test whether a listener is subscribed to a signal, using the `isSubscribed()` method which also takes the same parameter list.

```
if (isSubscribed(lengthSignalId, listener))
{
    ...
}
```

For completeness, there are methods for getting the list of signals that the component has subscribed to (`getLocalListenedSignals()`), and the list of listeners for a given signal (`getLocalSignalListeners()`). The former returns `std::vector<simsignal_t>`; the latter takes a signal ID (`simsignal_t`) and returns `std::vector<cIListener*>`.

The following example prints the number of listeners for each signal:

```
EV << "Signal listeners:\n";
std::vector<simsignal_t> signals = getLocalListenedSignals();
for (unsigned int i = 0; i < signals.size(); i++) {
    simsignal_t signalID = signals[i];
    std::vector<cIListener*> listeners = getLocalSignalListeners(signalID);
    EV << getSignalName(signalID) << ": " << listeners.size() << " signals\n";
}
```

Listeners

Listeners are objects that subclass from the `cIListener` class, which declares the following methods:

```
class cIListener
{
  public:
    virtual ~cIListener() {}
    virtual void receiveSignal(cComponent *src, simsignal_t id, bool b) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id, long l) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id, double d) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id, simtime_t t) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id, const char *s) = 0;
    virtual void receiveSignal(cComponent *src, simsignal_t id, cObject *obj) = 0;
    virtual void finish(cComponent *component, simsignal_t id) {}
    virtual void subscribedTo(cComponent *component, simsignal_t id) {}
    virtual void unsubscribedFrom(cComponent *component, simsignal_t id) {}
};
```

This class has a number of virtual methods:

- Several overloaded `receiveSignal()` methods, one for each data type. Whenever a signal is emitted (via `emit()`), the matching `receiveSignal()` methods of subscribed listeners are invoked.
- `finish()` is called by a component on its local listeners after the component's `finish()` method was called. If the listener is subscribed to multiple signals or at multiple components, the method will be called multiple times. Note that `finish()` methods in general are not invoked if the simulation terminates with an error, so this method is not a place for doing cleanup.
- `subscribedTo()`, `unsubscribedFrom()` are called when this listener object is subscribed/unsubscribed to (from) a signal. These methods give the opportunity for listeners to track whether and where they are subscribed. It is also OK for a listener to delete itself in the last statement of the `unsubscribedFrom()` method, but you must be sure that there are no other places the same listener is still subscribed.

Since `cIListener` has a large number of pure virtual methods, it is more convenient to subclass from `cListener`, a do-nothing implementation instead. It defines `finish()`, `subscribedTo()` and `unsubscribedFrom()` with an empty body, and the `receiveSignal()` methods with a bodies that throw a `"Data type not supported"` error. You can redefine the `receiveSignal()` method(s) whose data type you want to support, and signals emitted with other (unexpected) data types will result in an error instead of going unnoticed.

The order in which listeners will be notified is undefined (it is not necessarily the same order in which listeners were subscribed.)

Listener Life Cycle

When a component (module or channel) is deleted, it automatically unsubscribes (but does not delete) the listeners it has. When a module is deleted, it first unsubscribes all listeners from all modules and channels in its submodule tree before starting to recursively delete the modules and channels themselves.

When a listener is deleted, it must already be unsubscribed from all components at that point. If it is not unsubscribed, pointers to the dead listener object will be left in the components' listener lists, and the components will crash inside an `emit()` call, or when they try to invoke `unsubscribedFrom()` on the dead listener from their destructors. The `cIListener` class contains a subscription count, and prints a warning message when it is not zero in the destructor.

> **NOTE:** If your module has added listeners to other modules (e.g. the toplevel module), these listeners must be unsubscribed in the module destructor at latest. Remember to make sure the modules still exist before you call `unsubscribe()` on them, unless they are an ancestor of your module in the module tree.

4.14.3 Listening to Model Changes

In simulation models it is often useful to hold references to other modules, a connecting channel or other objects, or to cache information derived from the model topology. However, such pointers or data may become invalid when the model changes at runtime, and need to be updated or recalculated. The problem is how to get notification that something has changed in the model.

NOTE: Whenever you see a `cModule*`, `cChannel*`, `cGate*` or similar pointer kept as state in a simple module, you should think about how it will be kept up-to-date if the model changes at runtime.

The solution is, of course, signals. OMNeT++ has two built-in signals, `PRE_MODEL_CHANGE` and `POST_MODEL_CHANGE` (these macros are `simsignal_t` values, not names) that are emitted before and after each model change.

Pre/post model change notifications are emitted with data objects that carry the details of the change. The data classes are:

- `cPreModuleAddNotification` / `cPostModuleAddNotification`
- `cPreModuleDeleteNotification` / `cPostModuleDeleteNotification`
- `cPreModuleReparentNotification` / `cPostModuleReparentNotification`
- `cPreGateAddNotification` / `cPostGateAddNotification`
- `cPreGateDeleteNotification` / `cPostGateDeleteNotification`
- `cPreGateVectorResizeNotification` / `cPostGateVectorResizeNotification`
- `cPreGateConnectNotification` / `cPostGateConnectNotification`
- `cPreGateDisconnectNotification` / `cPostGateDisconnectNotification`
- `cPrePathCreateNotification` / `cPostPathCreateNotification`
- `cPrePathCutNotification` / `cPostPathCutNotification`
- `cPreParameterChangeNotification` / `cPostParameterChangeNotification`
- `cPreDisplayStringChangeNotification` / `cPostDisplayStringChangeNotification`

They all subclass from `cModelChangeNotification`, which is of course a `cObject`. Inside the listener, you can use `dynamic_cast<>` to figure out what notification arrived.

NOTE: Please look up these classes in the API documentation to see their data fields, when exactly they get fired, and what one needs to be careful about when using them.

An example listener that prints a message when a module is deleted:

```
class MyListener : public cListener
{
   ...
};

void MyListener::receiveSignal(cComponent *src, simsignal_t id, cObject *obj)
{
    if (dynamic_cast<cPreModuleDeleteNotification *>(obj))
    {
        cPreModuleDeleteNotification *data = (cPreModuleDeleteNotification *)obj;
        EV << "Module " << data->module->getFullPath() << " is about to be deleted\n"
    }
}
```

If you'd like to get notification about the deletion of any module, you need to install the listener on the system module:

```
simulation.getSystemModule()->subscribe(PRE_MODEL_CHANGE, listener);
```

NOTE: `PRE_MODEL_CHANGE` and `POST_MODEL_CHANGE` are fired on the module (or channel) affected by the change, and *not* on the module which executes the code that causes the change. For example, *pre-module-deleted* is fired on the module to be removed, and *post-module-deleted* is fired on its parent (because the original module no longer exists), and not on the module that contains the `deleteModule()` call.

NOTE: A listener will *not* receive *pre/post-module-deleted* notifications if the whole submodule tree that contains the subscription point is deleted. This is because compound module destructors begin by unsubscribing all modules/channels in the subtree before starting recursive deletion.

4.15 Signal-Based Statistics Recording

4.15.1 Motivation

One use of signals is to expose variables for result collection without telling where, how, and whether to record them. With this approach, modules only publish the variables, and the actual result recording takes place in listeners. Listeners may be added by the simulation framework (based on the configuration), or by other modules (for example by dedicated result collection modules).

The signals approach allows for several possibilities:

- Provides a controllable level of detail: in some simulation runs you may want to record all values as a time series, in other runs only record the mean, time average, minimum/-maximum value, standard deviation etc, and in yet other runs you may want to record the distribution as a histogram;
- Depending on the purpose of the simulation experiment, you may want to process the results before recording them, for example record a smoothed or filtered value, record the percentage of time the value is nonzero or over a threshold, record the sum of the values, etc.;
- You may want aggregate statistics, e.g. record the total number of packet drops or the average end-to-end delay for the whole network;
- You may want to record combined statistics, for example a drop percentage (drop count/total number of packets);
- You may want to ignore results generated during the warm-up period or during other transients.

With the signals approach the above goals can be fulfilled.

4.15.2 Declaring Statistics

Introduction

In order to record simulation results based on signals, one must add `@statistic` properties to the simple module's (or channel's) NED definition. A `@statistic` property defines the name of the statistic, which signal(s) are used as input, what processing steps are to be applied to them (e.g. smoothing, filtering, summing, differential quotient), and what properties are to be recorded (minimum, maximum, average, etc.) and in which form (vector, scalar, histogram). Record items can be marked optional, which lets you denote a *"default"* and a more comprehensive *"all"* result set to be recorded; the list of record items can be further tweaked from the configuration. One can also specify a descriptive name ("title") for the statistic, and also a measurement unit.

The following example declares a queue module with a queue length statistic:

```
simple Queue
{
    parameters:
        @statistic[queueLength](record=max,timeavg,vector?);
    gates:
        input in;
        output out;
}
```

As you can see, statistics are represented with indexed NED properties (see 3.12). The property name is always `statistic`, and the index (here, `queueLength`) is the name of the statistic. The property value, that is, everything inside the parentheses, carries hints and extra information for recording.

The above `@statistic` declaration assumes that module's C++ code emits the queue's updated length as signal `queueLength` whenever elements are inserted into the queue or are removed from it. By default, the maximum and the time average of the queue length will be recorded as scalars. One can also instruct the simulation (or parts of it) to record "all" results; this will turn on optional record items, those marked with a question mark, and then the queue lengths will also be recorded into an output vector.

> **NOTE:** The configuration lets you fine-tune the list of result items even beyond the `default` and `all` settings; see section 12.2.4.

In the above example, the signal to be recorded was taken from the statistic name. When that is not suitable, the `source` property key lets you specify a different signal as input for the statistic. The following example assumes that the C++ code emits a `qlen` signal, and declares a `queueLength` statistic based on that:

```
simple Queue
{
    parameters:
        @signal[qlen](type=int); // optional
        @statistic[queueLength](source=qlen; record=max,timeavg,vector?);
        ...
}
```

Note that beyond the `source=qlen` property key we have also added a signal declaration (`@signal` property) for the `qlen` signal. Declaring signals is currently optional and in fact `@signal` properties are currently ignored by the system, but it is a good practice nevertheless.

It is also possible to apply processing to a signal before recording it. Consider the following example:

```
@statistic[dropCount](source=count(drop); record=last,vector?);
```

This records the total number of packet drops as a scalar, and optionally the number of packets dropped in the function of time as a vector, provided the C++ code emits a `drop` signal every time a packet is dropped. The value and even the data type of the `drop` signal is indifferent, because only the number of emits will be counted. Here, `count()` is a *result filter*.

> **NOTE:** Starting from OMNeT++ 4.4, items containing parens (e.g. `count(drop)`) no longer need to be enclosed in quotation marks.

Another example:

```
@statistic[droppedBytes](source=sum(packetBytes(pkdrop)); record=last,
vector?);
```

This example assumes that the C++ code emits a `pkdrop` signal with a packet (`cPacket*` pointer) as a value. Based on that signal, it records the total number of bytes dropped (as a scalar, and optionally as a vector too). The `packetBytes()` filter extracts the number of bytes from each packet using `cPacket`'s `getByteLength()` method, and the `sum()` filter, well, sums them up.

Arithmetic expressions can also be used. For example, the following line computes the number of dropped bytes using the `packetBits()` filter.

```
@statistic[droppedBytes](source=sum(8*packetBits(pkdrop)); record=last,
vector?);
```

The source can also combine multiple signals in an arithmetic expression:

```
@statistic[dropRate](source=count(drop)/count(pk); record=last,vector?);
```

When multiple signals are used, a value arriving on either signal will result in one output value. The computation will use the last values of the other signals (sample-hold interpolation). One limitation regarding multiple signals is that the same signal cannot occur twice, because it would cause glitches in the output.

Record items may also be expressions and contain filters. For example, the statistic below is functionally equivalent to one of the above examples: it also computes and records as scalar and as vector the total number of bytes dropped, using a `cPacket*`-valued signal as input; however, some of the computations have been shifted into the recorder part.

```
@statistic[droppedBytes](source=packetBits(pkdrop); record=last(8*sum),
vector(8*sum)?);
```

Property Keys

The following keys are understood in `@statistic` properties:

source : Defines the input for the recorders (see `record=` key). When missing, the statistic name is taken as the signal name;

record : Contains a list of recording modes, separated by comma. Recording modes define how to record the source (see `source=` key).

title : A longer, descriptive name for the statistic signal; result visualization tools may use it as chart label, e.g. in the legend.

unit : Measurement unit of the values. This may also appear in charts.

interpolationmode : Defines how to interpolate signal values where needed (e.g. for drawing); possible values are `none`, `sample-hold`, `backward-sample-hold`, `linear`.

enum : Defines symbolic names for various integer signal values. The property value must be a string, containing *name=value* pairs separated by comma. Example: `"IDLE=1,BUSY=2,DOWN=3"`.

Available Filters and Recorders

The following table contains the list of predefined result filters. All filters in the table output a value for each input value.

Filter	Description
`count`	Computes and outputs the count of values received so far.
`sum`	Computes and outputs the sum of values received so far.
`min`	Computes and outputs the minimum of values received so far.
`max`	Computes and outputs the maximum of values received so far.
`mean`	Computes and outputs the average (sum / count) of values received so far.
`timeavg`	Regards the input values and their timestamps as a step function (sample-hold style), and computes and outputs its time average (integral divided by duration).
`constant0`	Outputs a constant 0 for each received value (independent of the value).
`constant1`	Outputs a constant 1 for each received value (independent of the value).
`packetBits`	Expects `cPacket` pointers as value, and outputs the bit length for each received one. Non-`cPacket` values are ignored.
`packetBytes`	Expects `cPacket` pointers as value, and outputs the byte length for each received one. Non-`cPacket` values are ignored.
`sumPerDuration`	For each value, computes the sum of values received so far, divides it by the duration, and outputs the result.
`removeRepeats`	Removes repeated values, i.e. discards values that are the same as the previous value.

The list of predefined result recorders:

Recorder	Description
`last`	Records the last value into an output scalar.
`count`	Records the count of the input values into an output scalar; functionally equivalent to `last(count)`

`sum`	Records the sum of the input values into an output scalar (or zero if there was none); functionally equivalent to `last(sum)`
`min`	Records the minimum of the input values into an output scalar (or positive infinity if there was none); functionally equivalent to `last(min)`
`max`	Records the maximum of the input values into an output scalar (or negative infinity if there was none); functionally equivalent to `last(max)`
`mean`	Records the mean of the input values into an output scalar (or NaN if there was none); functionally equivalent to `last(mean)`
`timeavg`	Regards the input values with their timestamps as a step function (sample-hold style), and records the time average of the input values into an output scalar; functionally equivalent to `last(timeavg)`
`stats`	Computes basic statistics (count, mean, std.dev, min, max) from the input values, and records them into the output scalar file as a statistic object.
`histogram`	Computes a histogram and basic statistics (count, mean, std.dev, min, max) from the input values, and records the reslut into the output scalar file as a histogram object.
`vector`	Records the input values with their timestamps into an output vector.

NOTE: You can have the list of available result filters and result recorders printed by executing the `opp_run -h resultfilters` and `opp_run -h resultrecorders` commands.

Naming and Attributes of Recorded Results

The names of recorded result items will be formed by concatenating the statistic name and the recording mode with a colon between them: "*<statisticName>*:*<recordingMode>*".

Thus, the following statistics

```
@statistic[dropRate](source=count(drop)/count(pk); record=last,vector?);
@statistic[droppedBytes](source=packetBytes(pkdrop); record=sum,vector(sum)?);
```

will produce the following scalars: `dropRate:last`, `droppedBytes:sum`, and the following vectors: `dropRate:vector`, `droppedBytes:vector(sum)`.

All property keys (except for `record`) are recorded as result attributes into the vector file or scalar file. The `title` property will be tweaked a little before recording: the recording mode will be added after a comma, otherwise all result items saved from the same statistic would have exactly the same name.

Example: "Dropped Bytes, sum", "Dropped Bytes, vector(sum)"

It is allowed to use other property keys as well, but they won't be interpreted by the OMNeT++ runtime or the result analysis tool.

Source and Record Expressions in Detail

To fully understand `source` and `record`, it will be useful to see how result recording is set up.

When a module or channel is created in the simulation, the OMNeT++ runtime examines the `@statistic` properties on its NED declaration, and adds listeners on the signals they mention as input. There are two kinds of listeners associated with result recording: *result filters* and *result recorders*. Result filters can be chained, and at the end of the chain there is always a recorder. So, there may be a recorder directly subscribed to a signal, or there may be a chain of one or more filters plus a recorder. Imagine it as a pipeline, or rather a "pipe tree", where the tree roots are signals, the leaves are result recorders, and the intermediate nodes are result filters.

Result filters typically perform some processing on the values they receive on their inputs (the previous filter in the chain or directly a signal), and propagate them to their output (chained filters and recorders). A filter may also swallow (i.e. not propagate) values. Recorders may write the received values into an output vector, or record output scalar(s) at the end of the simulation.

Many operations exist both in filter and recorder form. For example, the `sum` *filter* propagates the sum of values received on its input to its output; and the `sum` *recorder* only computes the the sum of received values in order to record it as an output scalar on simulation completion.

The next figure illustrates which filters and recorders are created and how they are connected for the following statistics:

```
@statistic[droppedBytes](source=8*packetBits(pkdrop); record=sum,vector(sum));
```

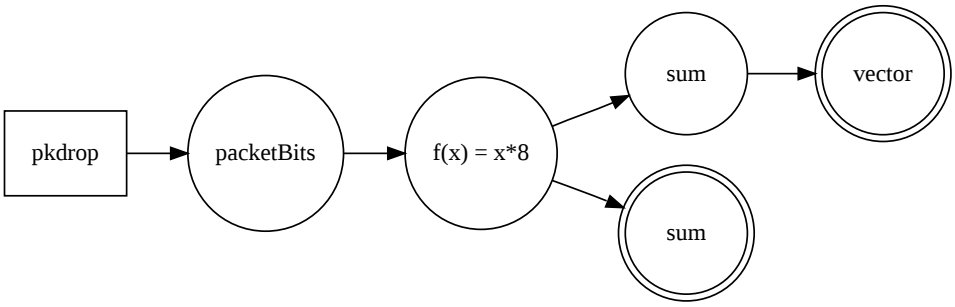


Figure 4.4: Result filters and recorders chained

HINT: To see how result filters and recorders have been set up for a particular simulation, run the simulation with the **`debug-statistics-recording`** configuration option, e.g. specify `-debug-statistics-recording=true` on the command line.

4.15.3 Statistics Recording for Dynamically Registered Signals

It is often convenient to have a module record statistics per session, per connection, per client, etc. One way of handling this use case is registering signals dynamically (e.g. `session1-jitter`, `session2-jitter`, ...), and setting up `@statistic`-style result recording on each.

The NED file would look like this:

```
@signal[session*-jitter](type=simtime_t); // note the wildcard
@statisticTemplate[sessionJitter](record=mean,stddev,vector?);
```

In the C++ code of the module, you need to register each new signal with `registerSignal()`, and in addition, tell OMNeT++ to set up statistics recording for it as described by the `@statisticTemplate` property. The latter can be achieved by calling `ev.addResultRecorders()`.

```
char signalName[32];
sprintf(signalName, "session%d-jitter", sessionNum);
simsignal_t signal = registerSignal(signalName);

char statisticName[32];
sprintf(statisticName, "session%d-jitter", sessionNum);
cProperty *statisticTemplate =
    getProperties()->get("statisticTemplate", "sessionJitter");
ev.addResultRecorders(this, signal, statisticName, statisticTemplate);
```

In the `@statisticTemplate` property, the `source` key will be ignored (because the signal given as parameter will be used as source). The actual name and index of property will also be ignored. (With `@statistic`, the index holds the result name, but here the name is explicitly specified in the statisticName parameter.)

When multiple signals are recorded using a common `@statisticTemplate` property, you'll want the titles of the recorded statistics to differ for each signal. This can be achieved by using dollar variables in the `title` key of `@statisticTemplate`. The following variables are available:

- `$name`: name of the statistic
- `$component`: component fullpath
- `$mode`: recording mode
- `$namePart[0-9]+`: given part of statistic name, when split along colons (:); numbering starts with 1

For example, if the statistic name is `"conn:host1-to-host4(3):bytesSent"`, and the title is `"bytes sent in connection $namePart2"`, it will become `"bytes sent in connection host1-to-host4(3)"`.

4.15.4 Adding Result Filters and Recorders Programmatically

As an alternative to `@statisticTemplate` and `ev.addResultRecorders()`, it is also possible to set up result recording programmatically, by creating and attaching result filters and recorders to the desired signals.

The following code example sets up recording to an output vector after removing duplicate values, and is essentially equivalent to the following `@statistic` line:

```
@statistic[queueLength](source=qlen; record=vector(removeRepeats);
                        title="Queue Length"; unit=packets);
```

The C++ code:

```
simsignal_t signal = registerSignal("qlen");

cResultFilter *removeRepeatsFilter =
    cResultFilterDescriptor::get("removeRepeats")->create();

cResultRecorder *vectorRecorder =
    cResultRecorderDescriptor::get("vector")->create();
opp_string_map *attrs = new opp_string_map;
(*attrs)["title"] = "Queue Length";
(*attrs)["unit"] = "packets";
vectorRecorder->init(this, "queueLength", "vector", NULL, attrs);

subscribe(signal, removeRepeatsFilter);
removeRepeatsFilter->addDelegate(vectorRecorder);
```

4.15.5 Emitting Signals

Emitting signals for statistical purposes does not differ much from emitting signals for any other purpose. Statistic signals are primarily expected to contain numeric values, so the overloaded `emit()` functions that take `long`, `double` and `simtime_t` are going to be the most useful ones.

Emitting with timestamp. The emitted values are associated with the current simulation time. At times it might be desirable to associate them with a different timestamp, in much the same way as the `recordWithTimestamp()` method of `cOutVector` (see 7.9.1) does. For example, assume that you want to emit a signal at the start of every successful wireless frame reception. However, whether any given frame reception is going to be successful can only be known after the reception has completed. Hence, values can only be emitted at reception completion, and need to be associated with past timestamps.

To emit a value with a different timestamp, an object containing a $(timestamp, value)$ pair needs to be filled in, and emitted using the `emit(simsignal_t, cObject *)` method. The class is called `cTimestampedValue`, and it simply has two public data members called `time` and `value`, with types `simtime_t` and `double`. It also has a convenience constructor taking these two values.

> **NOTE:** `cTimestampedValue` is not part of the signal mechanism. Instead, the result recording listeners provided by OMNeT++ have been written in a way so that they understand `cTimestampedValue`, and know how to handle it.

An example usage:

```
simtime_t frameReceptionStartTime = ...;
double receivePower = ...;
cTimestampedValue tmp(frameReceptionStartTime, receivePower);
emit(recvPowerSignal, &tmp);
```

If performance is critical, the `cTimestampedValue` object may be made a class member or a static variable to eliminate object construction/destruction time.[11]

Timestamps must be monotonically increasing.

[11] It is safe to use a static variable here because the simulation program is single-threaded, but ensure that there isn't a listener somewhere that would modify the same static variable during firing.

Emitting non-numeric values. Sometimes it is practical to have multi-purpose signals, or to retrofit an existing non-statistical signal so that it can be recorded as a result. For this reason, signals having non-numeric types (that is, `const char *` and `cObject *`) may also be recorded as results. Wherever such values need to be interpreted as numbers, the following rules are used by the built-in result recording listeners:

- Strings are recorded as 1.0, except for `NULL` which is recorded as 0.0;
- Objects that can be cast to `cITimestampedValue` are recorded using the `getSignalTime()` and `getSignalValue()` methods of the class;
- Other objects are recorded as 1.0, except for `NULL` pointers which are recorded as 0.0.

`cITimestampedValue` is a C++ interface that may be used as an additional base class for any class. It is declared like this:

```
class cITimestampedValue {
    public:
        virtual ~cITimestampedValue() {}
        virtual double getSignalValue(simsignal_t signalID) = 0;
        virtual simtime_t getSignalTime(simsignal_t signalID);
};
```

`getSignalValue()` is pure virtual (it must return some value), but `getSignalTime()` has a default implementation that returns the current simulation time. Note the `signalID` argument that allows the same class to serve multiple signals (i.e. to return different values for each).

4.15.6 Writing Result Filters and Recorders

You can define your own result filters and recorders in addition to the built-in ones. Similar to defining modules and new NED functions, you have to write the implementation in C++, and then register it with a registration macro to let OMNeT++ know about it. The new result filter or recorder can then be used in the `source=` and `record=` attributes of `@statistic` properties just like the built-in ones.

Result filters must be subclassed from `cResultFilter` or from one of its more specific subclasses `cNumericResultFilter` and `cObjectResultFilter`. The new result filter class needs to be registered using the `Register_ResultFilter(NAME, CLASSNAME)` macro.

Similarly, a result recorder must subclass from the `cResultRecorder` or the more specific `cNumericResultRecorder` class, and be registered using the `Register_ResultRecorder(NAME, CLASSNAME)` macro.

An example result filter implementation from the simulation runtime:

```
/**
 * Filter that outputs the sum of signal values divided by the measurement
 * interval (simtime minus warmup period).
 */
class SumPerDurationFilter : public cNumericResultFilter
{
    protected:
        double sum;
    protected:
```

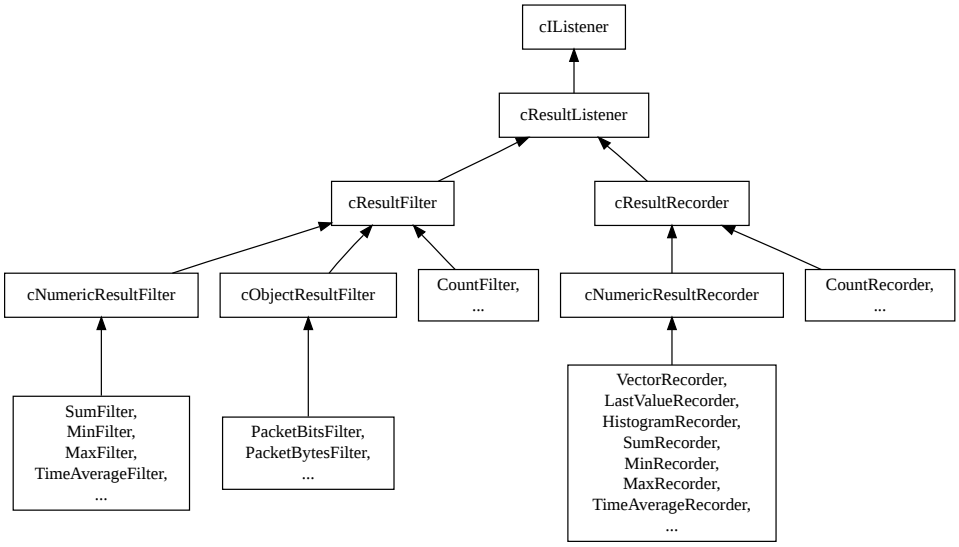


Figure 4.5: Inheritance of result filter and recorder classes

```
        virtual bool process(simtime_t& t, double& value);
    public:
        SumPerDurationFilter() {sum = 0;}
};

Register_ResultFilter("sumPerDuration", SumPerDurationFilter);

bool SumPerDurationFilter::process(simtime_t& t, double& value)
{
    sum += value;
    value = sum / (simTime() - simulation.getWarmupPeriod());
    return true;
}
```

Chapter 5

Messages and Packets

5.1 Overview

Messages are a central concept in OMNeT++. In the model, message objects represent events, packets, commands, jobs, customers or other kinds of entities, depending on the model domain.

Messages are represented with the `cMessage` class and its subclass `cPacket`. `cPacket` is used for network packets (frames, datagrams, transport packets, etc.) in a communication network, and `cMessage` is used for everything else. Users are free to subclass both `cMessage` and `cPacket` to create new types and to add data.

`cMessage` has the following fields; some are used by the simulation kernel, and others are provided for the convenience of the simulation programmer:

- The *name* field is a string (`const char *`), which can be freely used by the simulation programmer. The message name is displayed at many places in the graphical runtime interface, so it is generally useful to choose a descriptive name. Message name is inherited from `cObject` (see section 7.1.1).

- *Message kind* is an integer field. Some negative values are reserved by the simulation library, but zero and positive values can be freely used in the model for any purpose. Message kind is typically used to carry a value that conveys the role, type, category or identity of the message.

- The *scheduling priority* field is used by the simulation kernel to determine the delivery order of messages that have the same arrival time values. This field is rarely used in practice.

- The *send time*, *arrival time*, *source module*, *source gate*, *destination module*, *destination gate* fields store information about the message's last sending or scheduling, and should not be modified from the model. These fields are primarily used internally by the simulation kernel while the message is in the future events set (FES), but the information is still in the message object when the message is delivered to a module.

- *Time stamp* (not to be confused with *arrival time*) is a utility field, which the programmer can freely use for any purpose. The time stamp is not examined or changed by the simulation kernel at all.

- The *parameter list*, *control info* and *context pointer* fields make some simulation tasks easier to program, and they will be discussed later.

The `cPacket` class extends `cMessage` with fields that are useful for representing network packets:

- The *packet length* field represents the length of the packet in bits. It is used by the simulation kernel to compute the transmission duration when a packet travels through a connection that has an assigned data rate, and also for error modeling on channels with a nonzero bit error rate.
- The *encapsulated packet* field helps modeling protocol layers by supporting the concept of encapsulation and decapsulation.
- The *bit error flag* field carries the result of error modelling after the packet is sent through a channel that has a nonzero packet error rate (PER) or bit error rate (BER). It is up to the receiver to examine this flag after having received the packet, and to act upon it.
- The *duration* field carries the transmission duration after the packet was sent through a channel with a data rate.
- The *is-reception-start* flag tells whether this packet represents the start or the end of the reception after the packet travelled through a channel with a data rate. This flag is controlled by the *deliver-on-reception-start* flag of the receiving gate.

5.2 The cMessage Class

5.2.1 Basic Usage

The `cMessage` constructor accepts an *object name* and a *message kind*, both optional:

```
cMessage(const char *name=NULL, short kind=0);
```

Descriptive message names can be very useful when tracing, debugging or demonstrating the simulation, so it is recommended to use them. Message kind is usually initialized with a symbolic constant (e.g. an *enum* value) which signals what the message object represents. Only positive values and zero can be used – negative values are reserved for use by the simulation kernel.

The following lines show some examples of message creation:

```
cMessage *msg1 = new cMessage();
cMessage *msg2 = new cMessage("timeout");
cMessage *msg3 = new cMessage("timeout", KIND_TIMEOUT);
```

Once a message has been created, its basic data members can be set with the following methods:

```
void setName(const char *name);
void setKind(short k);
void setTimestamp();
void setTimestamp(simtime_t t);
void setSchedulingPriority(short p);
```

The argument-less `setTimeStamp()` method is equivalent to `setTimeStamp(simTime())`.

The corresponding getter methods are:

```
const char *getName() const;
short getKind() const;
simtime_t getTimestamp() const;
short getSchedulingPriority() const;
```

The `getName()`/`setName()` methods are inherited from a generic base class in the simulation library, `cNamedObject`.

Two more interesting methods:

```
bool isPacket() const;
simtime_t getCreationTime() const;
```

The `isPacket()` method returns `true` if the particular message object is a subclass of `cPacket`, and `false` otherwise. As `isPacket()` is implemented as a virtual function that just contains a `return false` or a `return true` statement, it might be faster than calling `dynamic_cast<cPacket*>`.

The `getCreationTime()` method returns the creation time of the message. It is worthwhile to mention that with cloned messages (see `dup()` later), the creation time of the original message is returned and not the time of the cloning operation. This is particularly useful when modeling communication protocols, because many protocols clone the transmitted packages to be able to do retransmissions and/or segmentation/reassembly.

5.2.2 Duplicating Messages

It is often necessary to duplicate a message or a packet, for example, to send one and keep a copy. Duplication can be done in the same way as for any other OMNeT++ object:

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

The resulting message (or packet) will be an exact copy of the original including message parameters and encapsulated messages, except for the message ID field. The creation time field is also copied, so for cloned messages `getCreationTime()` will return the creation time of the original, not the time of the cloning operation. [1]

If you subclass from `cMessage` or `cPacket`, you need to reimplement `dup()`; the recommended implementation is to delegate to the copy constructor of the new class:

```
class FooMessage : public cMessage {
  public:
    FooMessage(const FooMessage& other) {...}
    virtual FooMessage *dup() const {return new FooMessage(*this);}
    ...
};
```

For generated classes (chapter 6), this is taken care of automatically.

[1] Note, however, that the simulation library may delay the duplication of the encapsulated message until it is really needed; see section 5.4.5.

5.2.3 Message IDs

Every message object has a unique numeric *message ID*. It is normally used for identifying the message in a recorded event log file, but may occasionally be useful for other purposes as well. If you clone a message (`msg->dup()`), the clone will have a different ID.

There is also another ID called *tree ID*. A tree ID starts out with the value of the message ID; however, if you clone a message, the clone will retain the tree ID of the original. Thus, messages that have the same tree ID have been created by cloning the same original message or its clones (with one exception, the original message). The size of `long` is usually enough so that IDs remain unique during a single simulation run (i.e. the counter does not wrap).

The methods for obtaining message IDs:

```
long getId() const;
long getTreeId() const;
```

5.2.4 Control Info

One of the main application areas of OMNeT++ is the simulation of telecommunication networks. Here, protocol layers are usually implemented as modules which exchange packets. Packets themselves are represented by messages subclassed from `cPacket`.

However, communication between protocol layers requires sending additional information to be attached to packets. For example, a TCP implementation sending down a TCP packet to IP will want to specify the destination IP address and possibly other parameters. When IP passes up a packet to TCP after decapsulation from the IP header, it will want to let TCP know at least the source IP address.

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

```
void setControlInfo(cObject *controlInfo);
cObject *getControlInfo() const;
cObject *removeControlInfo();
```

When a "command" is associated with the message sending (such as TCP OPEN, SEND, CLOSE, etc), the message kind field (`getKind()`, `setKind()` methods of `cMessage`) should carry the command code. When the command doesn't involve a data packet (e.g. TCP CLOSE command), a dummy packet (empty `cMessage`) can be sent.

An object set as control info via `setControlInfo()` will be owned by the message object. When the message is deallocated, the control info object is deleted as well.

5.2.5 Information About the Last Arrival

The following methods return the sending and arrival times that correspond to the last sending of the message.

```
simtime_t getSendingTime() const;
simtime_t getArrivalTime() const;
```

The following methods can be used to determine where the message came from and which gate it arrived on (or will arrive if it is currently scheduled or under way.) There are two sets of methods, one returning module/gate Ids, and the other returning pointers.

```
int getSenderModuleId() const;
int getSenderGateId() const;
int getArrivalModuleId() const;
int getArrivalGateId() const;
cModule *getSenderModule() const;
cGate *getSenderGate() const;
cModule *getArrivalModule() const;
cGate *getArrivalGate() const;
```

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

```
bool arrivedOn(int gateId) const;
bool arrivedOn(const char *gatename) const;
bool arrivedOn(const char *gatename, int gateindex) const;
```

5.2.6 Display String

Display strings affect the message's visualization on animating user interfaces, e.g. Tkenv. Message objects do not store a display string by default, but contain a `getDisplayString()` method that can be overridden in subclasses to return a desired string. The method:

```
const char *getDisplayString() const;
```

See chapter 8 for more information on display strings.

5.3 Self-Messages

5.3.1 Using a Message as Self-Message

Messages are often used to represent events internal to a module, such as a periodically firing timer to represent expiry of a timeout. A message is termed *self-message* when it is used in such a scenario – otherwise self-messages are normal messages of class `cMessage` or a class derived from it.

When a message is delivered to a module by the simulation kernel, you can call the `isSelfMessage()` method to determine if it is a self-message; it other words, if it was scheduled with `scheduleAt()` or was sent with one of the `send...()` methods. The `isScheduled()` method returns true if the message is currently scheduled. A scheduled message can also be cancelled (`cancelEvent()`).

```
bool isSelfMessage() const;
bool isScheduled() const;
```

The methods `getSendingTime()` / `getArrivalTime()` are also useful with self-messages: they return the time the message was scheduled and arrived (or will arrive; while the message is scheduled, arrival time is the time it will be delivered to the module).

5.3.2 Context Pointer

`cMessage` contains a *context pointer* of type `void*`, which can be accessed by the following functions:

```
void setContextPointer(void *p);
void *getContextPointer() const;
```

The context pointer can be used for any purpose by the simulation programmer. It is not used by the simulation kernel, and it is treated as a mere pointer (no memory management is done on it).

Intended purpose: a module which schedules several self-messages (timers) will need to identify a self-message when it arrives back to the module, ie. the module will have to determine which timer went off and what to do then. The context pointer can be made to point at a data structure kept by the module which can carry enough "context" information about the event.

5.4 The cPacket Class

5.4.1 Basic Usage

The `cPacket` constructor is similar to the `cMessage` constructor, but it accepts an additional *bit length* argument:

```
cPacket(const char *name=NULL, short kind=0, int64 bitLength=0);
```

The most important field `cPacket` has over `cMessage` is the message length. This field is kept in bits, but it can also be set/get in bytes. If the bit length is not a multiple of eight, the `getByteLength()` method will round it up.

```
void setBitLength(int64 l);
void setByteLength(int64 l);
void addBitLength(int64 delta);
void addByteLength(int64 delta);
int64 getBitLength() const;
int64 getByteLength() const;
```

Another extra field is the bit error flag. It can be accessed with the following methods:

```
void setBitError(bool e);
bool hasBitError() const;
```

5.4.2 Identifying the Protocol

In OMNeT++ protocol models, the protocol type is usually represented in the message subclass. For example, instances of class `IPv6Datagram` represent IPv6 datagrams and `EthernetFrame` represents Ethernet frames. The C++ `dynamic_cast` operator can be used to determine if a message object is of a specific protocol.

An example:

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

```
{
    IPv6Datagram *datagram = (IPv6Datagram *)msg;
    ...
}
```

5.4.3 Information About the Last Transmission

When a packet has been received, some information can be obtained about the transmission, namely the *transmission duration* and the *is-reception-start* flag. They are returned by the following methods:

```
simtime_t getDuration() const;
bool isReceptionStart() const;
```

5.4.4 Encapsulating Packets

It is often necessary to encapsulate a packet into another when you are modeling layered protocols of computer networks.

The following `cPacket` methods are associated with encapsulation:

```
void encapsulate(cPacket *packet);
cPacket *decapsulate();
cPacket *getEncapsulatedPacket() const;
```

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

A packet can only hold one encapsulated packet at a time; the second `encapsulate()` call will result in an error. It is also an error if the packet to be encapsulated is not owned by the module.

You can get back the encapsulated packet by calling `decapsulate()`. `decapsulate()` will decrease the length of the packet accordingly, except if it was zero. If the length would become negative, an error occurs.

The `getEncapsulatedPacket()` function returns a pointer to the encapsulated packet, or `NULL` if no packet is encapsulated.

Example usage:

```
cPacket *data = new cPacket("data");
data->setByteLength(1024);

UDPPacket *udp = new UDPPacket("udp"); // subclassed from cPacket
udp->setByteLength(8);

udp->encapsulate(data);
ev << udp->getByteLength() << endl; // --> 8+1024 = 1032
```

And the corresponding decapsulation code:

```
cPacket *payload = udp->decapsulate();
```

5.4.5 Reference Counting

Since the 3.2 release, OMNeT++ implements reference counting of encapsulated packets, meaning that if you `dup()` a packet that contains an encapsulated packet, then the encapsulated packet will not be duplicated, only a reference count incremented. Duplication of the encapsulated packet is deferred until `decapsulate()` actually gets called. If the outer packet is deleted without its `decapsulate()` method ever being called, then the reference count of the encapsulated packet is simply decremented. The encapsulated packet is deleted when its reference count reaches zero.

Reference counting can significantly improve performance, especially in LAN and wireless scenarios. For example, in the simulation of a broadcast LAN or WLAN, the IP, TCP and higher layer packets won't be duplicated (and then discarded without being used) if the MAC address doesn't match in the first place.

The reference counting mechanism works transparently. However, there is one implication: **one must not change anything in a packet that is encapsulated into another!** That is, `getEncapsulatedPacket()` should be viewed as if it returned a pointer to a read-only object (it returns a `const` pointer indeed), for quite obvious reasons: the encapsulated packet may be shared between several packets, and any change would affect those other packets as well.

5.4.6 Encapsulating Several Packets

The `cPacket` class does not directly support encapsulating more than one packet, but you can subclass `cPacket` or `cMessage` to add the necessary functionality. (It is recommended that you use the message definition syntax that will be described in chapter 6 – it can spare you some work.)

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

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

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

You will also have to provide an `operator=()` method to make sure your message objects can be copied and duplicated properly – this is something often needed in simulations (think of broadcasts and retransmissions!). Section 7.12 contains more about the things you need to take care of when deriving new classes.

5.5 Attaching Parameters and Objects

If you want to add parameters or objects to a message, the preferred way to do that is via message definitions, described in chapter 6.

5.5.1 Attaching Objects

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

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

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

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

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

5.5.2 Attaching Parameters

The preferred way of extending messages with new data fields is to use message definitions (see chapter 6).

The old, deprecated way of adding new fields to messages is via attaching `cMsgPar` objects. There are several downsides of this approach, the worst being large memory and execution time overhead. `cMsgPar`'s are heavy-weight and fairly complex objects themselves. It has been reported that using `cMsgPar` message parameters might account for a large part of execution time, sometimes as much as 80%. Using `cMsgPar` is also error-prone because `cMsgPar` objects have to be added dynamically and individually to each message object. In contrast, subclassing benefits from static type checking: if you mistype the name of a field in the C++ code, the compiler can detect the mistake.

If you still need `cMsgPar`s for some reason, here is a short summary. At the sender side you can add a new named parameter to the message with the `addPar()` member function, then set its value with one of the methods `setBoolValue()`, `setLongValue()`, `setStringValue()`, `setDoubleValue()`, `setPointerValue()`, `setObjectValue()`, and `setXMLValue()`. There are also overloaded assignment operators for the corresponding C/C++ types.

At the receiver side, you can look up the parameter object on the message by name and obtain a reference to it with the `par()` member function. `hasPar()` can be used to check first whether the message object has a parameter object with the given name. Then the value can be read with the methods `boolValue()`, `longValue()`, `stringValue()`, `doubleValue()`, `pointerValue()`, `objectValue()`, `xmlValue()`, or by using the provided overloaded type cast operators.

Example usage:

```
msg->addPar("destAddr");
msg->par("destAddr").setLongValue(168);
...
long destAddr = msg->par("destAddr").longValue();
```

Or, using overloaded operators:

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

Chapter 6

Message Definitions

6.1 Introduction

In practice, you will need to add various fields to `cMessage` or `cPacket` to make them useful. For example, if you are modelling packets in communication networks, you need to have a way to store protocol header fields in packets. Since the simulation library is written in C++, the natural way of extending `cMessage`/`cPacket` is via subclassing them. 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 offer you a compact syntax to describe message contents, and the corresponding C++ code is automatically generated from the definitions. A common complaint about code generators in general is lack of flexibility: if you have a different idea how the generated code should look, there is little you can do about it. OMNeT++, however, allows you to extensively customize the generated class. Even if you need to heavily customize the generated class, message definitions still save you a great deal of manual work.

6.1.1 The First Message Class

Let us begin with a simple example. Suppose that you need a packet class that carries source and destination addresses as well as a hop count. You may then write a `MyPacket.msg` file with the following contents:

```
packet MyPacket
{
    int srcAddress;
    int destAddress;
    int remainingHops = 32;
};
```

It is the task of the *message compiler* is to generate C++ classes you can use from your models. The message compiler is normally invoked automatically for your `.msg` files during build.

When the message compiler processes `MyPacket.msg`, it creates the following files: `MyPacket_m.h` and `MyPacket_m.cc`. The generated `MyPacket_m.h` will contain the following class declaration:

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

In your C++ files, you can use the `MyPacket` class by including the generated header file:

```
#include "MyPacket_m.h"

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

The `MyPacket_m.cc` file will contain implementation of the generated `MyPacket` class as well as "reflection" code that allows you to inspect these data structures in the Tkenv GUI. The `MyPacket_m.cc` file should be compiled and linked into your simulation; this is normally taken care of automatically.

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

6.2 Messages and Packets

6.2.1 Defining Messages and Packets

Message and packet contents can be defined in a syntax resembling C structs. The keyword can be **`message`** or **`packet`**; they cause the generated C++ class to be derived from `cMessage` and `cPacket`, respectively. (Further keywords, **`class`** and **`struct`**, will be covered later.)

An example packet definition:

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

Saving the above code into a `FooPacket.msg` file and processing it with the message compiler, `opp_msgc`, will produce the files `FooPacket_m.h` and `FooPacket_m.cc`. The header file will contain the declaration of the generated C++ class.

The generated class will have a constructor that optionally accepts object name and message kind, and also a copy constructor. An assignment operator (`operator=()`) and cloning method (`dup()`) will also be generated.

```
class FooPacket : public cPacket
{
  public:
    FooPacket(const char *name=NULL, int kind=0);
    FooPacket(const FooPacket& other);
    FooPacket& operator=(const FooPacket& other);
```

```
    virtual FooPacket *dup() const;
    ...
```

For each field in the above description, the generated class will have a protected data member, and a public getter and setter method. The names of the methods will begin with `get` and `set`, followed by the field name with its first letter converted to uppercase. Thus, `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.

String fields can also be declared:

```
packet HttpRequestMessage
{
    string method; // "GET", "POST", etc.
    string resource;
};
```

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

```
virtual const char *getMethod() const;
virtual void setMethod(const char *method);
virtual const char *getResource() const;
virtual void setResource(const char *resource);
```

The generated object will have its own copy of the string, so it not only stores the `const char*` pointer.

6.2.2 Field Data Types

Data types for fields are not limited to **`int`** and **`bool`**. You can use several C/C++ and other data types:

- logical: **`bool`**
- integral types: **`char`**, **`short`**, **`int`**, **`long`**; and their unsigned versions **`unsigned char`**, **`unsigned short`**, **`unsigned int`**, **`unsigned long`**
- floating-point types: **`float`**, **`double`**
- C99-style fixed-size integral types: **`int8_t`**, **`int16_t`**, **`int32_t`**, **`int64_t`**; and their unsigned versions **`uint8_t`**, **`uint16_t`**, **`uint32_t`**, **`uint64_t`**; [1]
- OMNeT++ simulation time: **`simtime_t`**

[1] These type names are accepted without the `_t` suffix as well, but you are responsible to ensure that the generated code compiles, i.e. the shortened type names must be defined in a header file you include.

- **string**. Getters and setters use the `const char*` data type; `NULL` is not allowed. The object will store a copy of the string, not just the pointer.
- structs and classes, defined in message files or elsewhere (see in later sections 6.2.7 and 6.6)
- typedef'd names declared in C++ and announced to the message compiler)

Numeric fields are initialized to zero, booleans to `false`, and string fields to empty string.

6.2.3 Initial Values

You can specify initial values for fields. Examples:

```
packet RequestPacket
{
    int version = HTTP_VERSION;
    string method = "GET";
    string resource = "/";
    int maxBytes = 100*1024*1024; // 100MiB
    bool keepAlive = true;
};
```

As you can see, macros and expressions are also accepted as initalizer values. The message compiler does not check the syntax of the values, it only copies them into the generated C++ file; so if there is an error in them, it will be reported by the C++ compiler.

Field initialization statements will be placed into the constructor of the generated class.

6.2.4 Enums

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

Example:

```
packet FooPacket
{
    int payloadType @enum(PayloadType);
};
```

The enum itself has to be declared separately. An enum is declared with the **enum** keyword, using the following syntax:

```
enum PayloadType
{
   NONE = 0;
   UDP = 1;
   TCP = 2;
   SCTP = 3;
};
```

Enum values need to be unique.

The message compiler translates an enum into 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.

If the enum to be associated with a field comes from a different message file, then the enum must be announced and its generated header file be included. An example:

```
cplusplus {{
#include "PayloadType_m.h"
}}

enum PayloadType;

packet FooPacket
{
    int payloadType @enum(PayloadType);
};
```

6.2.5 Fixed-Size Arrays

You can specify fixed size arrays:

```
packet SourceRoutedPacket
{
    int 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.

6.2.6 Variable-Size Arrays

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

```
packet SourceRoutedPacket
{
    int 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 `set...ArraySize()` with a nonzero argument before you can start filling array elements.

6.2.7 Classes and Structs as Fields

In addition to primitive types, you can also use other types (classes, structs, typedefs, etc.) as fields. For example, if you have a C++ type called `IPAddress`, you can write the following:

```
packet IPPacket
{
    int version = 4;
    IPAddress src;
    IPAddress dest;
};
```

The `IPAddress` type must be known to the message compiler, and also at compile time to the C++ compiler; section 6.6 will describe how to achieve that.

The generated class will contain `IPAddress` data members (that is, not pointers to `IPAddress` objects), and the following getter and setter methods will be generated for them:

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

virtual IPAddress& getDest();
virtual const IPAddress& getDest() const;
virtual void setDest(const IPAddress& dest);
```

6.2.8 Pointer Fields

Pointer fields where the setters and the destructor would delete the previous value are not supported yet. However, there are workarounds, as described below.

You can create a typedef for the pointer and use the typedef name as field type. Then you'll get a plain pointer field where neither the setter nor the destructor deletes the old value (which is a likely memory leak).

Example (section 6.6 will explain the details):

```
cplusplus {{ typedef Foo *FooPtr; }} // C++ typedef
class noncobject FooPtr; // announcement for the message compiler

packet Bar
{
    FooPtr fooPtr;  // leaky pointer field
};
```

Then you can customize the class via C++ inheritance and reimplement the setter methods in C++, inserting the missing `delete` statements. Customization via C++ inheritance will be described in section 6.7.2.

6.2.9 Inheritance

By default, messages are subclassed from `cMessage` or `cPacket`. However, you can explicitly specify the base class using the **`extends`** keyword (only single inheritance is supported):

```
packet Ieee80211DataFrame extends Ieee80211Frame
{
    ...
};
```

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

```
// generated C++
class Ieee80211DataFrame : public Ieee80211Frame {
    ...
};
```

6.2.10 Assignment of Inherited Fields

Message definitions allow you to change the initial value of a field defined in an ancestor type. The syntax is similar to that of a field definition with initial value, only the data type is missing.

An example:

```
packet Ieee80211Frame
{
    int frameType;
    ...
};

packet Ieee80211DataFrame extends Ieee80211Frame
{
    frameType = DATA_FRAME;  // assignment of inherited field
    ...
};
```

It may seem like the message compiler would need the definition of the base class to check the definition of the field being assigned. However, it is not the case. The message compiler trusts that such field exists; or rather, it leaves the check to the C++ compiler.

What the message compiler actually does is derives a setter method name from the field name, and generates a call to it into the constructor. Thus, the generated constructor for the above packet type would be something like this:

```
Ieee80211DataFrame::Ieee80211DataFrame(const char *name, int kind) :
    Ieee80211Frame(name, kind)
{
    this->setFrameType(DATA_FRAME);
    ...
}
```

This implementation also lets you initialize `cMessage` / `cPacket` fields such as message kind or packet length:

```
packet UDPPacket
{
    byteLength = 16;  // results in 'setByteLength(16);' being placed into ctor
};
```

6.3 Classes

Until now we have only seen message and packet descriptions, which generate classes derived from `cMessage` or `cPacket`. However, it is also useful to be able to generate classes and structs, for building blocks for messages, as control info objects (see `cMessage`'s `setControlInfo()` and for other purposes. This section covers classes; structs will be described 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 / packet`**. The base class can be specified with the **`extends`** keyword, and defaults to `cObject`.

> **NOTE:** `cObject` has no data members. It only defines virtual methods, so the only overhead would be the *vptr*; however, the generated class already has a vptr because the generated methods are also virtual. In other words, `cObject` adds zero overhead to the generated class, and there is no reason not to always use it as base class.

Examples:

```
class TCPCommand  // same as "extends cObject"
{
    ...
};

class TCPOpenCommand extends TCPCommand
{
    ...
};
```

The generated code:

```
// generated C++
class TCPCommand : public cObject
{
    ...
};

class TCPOpenCommand : public TCPCommand
{
    ...
};
```

6.4 Structs

You can define C-style structs to be used as fields in message classes, "C-style" meaning "containing only data and no methods" (in contrast to C++ where a struct is just a class with

a different default member visibility.)

The syntax is similar to that of defining messages:

```
struct Place
{
    int type;
    string description;
    double coords[3];
};
```

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

```
// generated C++
struct Place
{
    int type;
    opp_string description; // minimal string class that wraps a const char*
    double coords[3];
};
```

Note that **string** fields are generated with the `opp_string` C++ type, which is a minimalistic string class that wraps `const char*` and takes care of allocation/deallocation. It was chosen instead of `std::string` because of its significantly smaller memory footprint (the sizeof of `opp_string` is the same as that of a `const char*` pointer).

Inheritance is supported for structs:

```
struct Base
{
    ...
};

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

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

- variable-size arrays are not supported;
- customization via inheritance and **abstract** fields (see later in 6.7.2) cannot be used;
- cannot have classes subclassed from `cOwnedObject` as fields, because structs cannot be owners.

6.5 Literal C++ Blocks

It is possible to have C++ code placed directly into the generated code, more precisely, into the generated header file. This is done with the **cplusplus** keyword and a double curly braces. As we'll see in later sections, **cplusplus** blocks are customarily used to insert `#include` directives, typedefs, `#define` macros and other elements into the generated header.

Example:

```
cplusplus {{
#include <vector>
#include "foo.h"
#define FOO_VERSION 4
typedef std::vector<int> IntVector;
}}
```

The message compiler does not try to make sense of the text in the body of the `cplusplus` block, it just simply copies it into the generated header file.

6.6 Using C++ Types

The message compile only knows about the types defined within the same msg file, and the built-in types. To be able to use other types, for example for fields or as base class, you need to do two things:

1. Let the message compiler know about the type by announcing it; and
2. Make sure its C++ declaration will be available at compile time

The next two sections describe how to do each.

6.6.1 Announcing Types to the Message Compiler

If you want to use a C++ type (a class, struct or typedef) not declared with the message syntax in the same file, you have to announce those types to the message compiler.

Type annoucements have a similar syntax to those in C++:

```
struct Point;
class PrioQueue;  // implies it is derived from cOwnedObject! see below
message TimeoutMessage;
packet TCPSegment;
```

However, with the **`class`** keyword, the message compiler needs to know the whether the class is derived (directly or indirectly) from `cOwnedObject`, `cNamedObject`, `cObject` or none of the above, because it affects code generation. The ancestor class can be declared with the **`extends`** keyword, like this:

```
class IPAddress extends void;  // does not extend any "interesting" class
class ModulePtr extends void;  // ditto
class IntVector extends void;  // ditto
class IPCtlInfo extends cObject;
class FooOption extends cNamedObject;
class PrioQueue extends cOwnedObject;
class IPAddrExt extends IPAddress;  // also OK: IPAddress has been announced
```

An alternative to `extends void` is the **`noncobject`** modifier:

```
class noncobject IPAddress; // same as "extends void"
```

By default, that is, when **extends** is missing, it is assumed that the class is derived from cOwnedObject. Thus, the following two announcements are equivalent:

```
class PrioQueue;
class PrioQueue extends cOwnedObject;
```

NOTE: Notice that this default is inconsistent with the default base class for generating classes, which is cObject (see 6.3). The reason why type announcements assume cOwnedObject is that it is safer: a mistake will surface in the form of a compile error and will not remain hidden until it causes some obscure runtime error.

6.6.2 Making the C++ Declarations Available

In addition to announcing types to the message compiler, you also have to make sure their C++ declarations will be available at compile time so that the generated code will actually compile. This can be achieved with **cplusplus** blocks that let you inject includes, typedefs, class/struct declarations, etc. into the generated header file:

```
cplusplus {{
#include "IPAddress.h"
typedef std::vector<int> IntVector;
}}
```

You need a **cplusplus** block even if the desired types are defined in a (different) message file, to include the generated header file. It is currently not supported to import types from other message files directly. Example:

```
cplusplus {{
#include "TCPSegment_m.h"  // make types defined in TCPSegment.msg available
                           // for the C++ compiler
}}
```

6.6.3 Putting it Together

Suppose you have header files and message files that define various types:

```
// IPAddress.h
class IPAddress {
   ...
};
```

```
// Location.h
struct Location {
    double lon;
    double lat;
};
```

```
// AppPacket.msg
packet AppPacket {
   ...
}
```

To be able to use the above types in a message definition (and two more, an `IntVector` and a module pointer), the message file should contain the following lines:

```
cplusplus {{
#include <vector>
#include "IPAddress.h"
#include "Location.h"
#include "AppPacket_m.h"
typedef std::vector<int> IntVector;
typedef cModule *ModulePtr;
}};

class noncobject IPAddress;
struct Location;
packet AppPacket;
class noncobject IntVector;
class noncobject ModulePtr;

packet AppPacketExt extends AppPacket {
    IPAddress destAddress;
    Location senderLocation;
    IntVector data;
    ModulePtr originatingModule;
}
```

6.7 Customizing the Generated Class

6.7.1 Customizing Method Names

The names and some other properties of generated methods can be influenced with metadata annotations (properties).

The names of the getter and setter methods can be changed with the `@getter` and `@setter` properties. For variable-size array fields, the names of array size getter and setter methods can be changed with `@sizeGetter` and `@sizeSetter`.

In addition, the data type for the array size (by default `unsigned int`) can be changed with `@sizetype` property.

Consider the following example:

```
packet IPPacket {
    int ttl @getter(getTTL) @setter(setTTL);
    Option options[] @sizeGetter(getNumOptions)
                     @sizeSetter(setNumOptions)
                     @sizetype(short);
}
```

The generated class would have the following methods (note the differences from the default names `getTtl()`, `setTtl()`, `getOptions()`, `setOptions()`, `getOptionsArraySize()`, `getOptionsArraySize()`; also note that indices and array sizes are now `short`):

```
virtual int getTTL() const;
virtual void setTTL(int ttl);
```

```
virtual const Option& getOption(short k) const;
virtual void setOption(short k, const Option& option);
virtual short getNumOptions() const;
virtual void setNumOptions(short n);
```

In some older simulation models you may also see the use of the `@omitGetVerb` class property. This property tells the message compiler to generate getter methods without the "get" prefix, e.g. for a `sourceAddress` field it would generate a `sourceAddress()` method instead of the default `getSourceAddress()`. It is not recommended to use `@omitGetVerb` in new models, because it is inconsistent with the accepted naming convention.

6.7.2 Customizing the Class via Inheritance

Sometimes you need the generated code to do something more or do something differently than the version generated by the message compiler. For example, when setting an 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)
{
    addByteLength(payloadLength - this->payloadLength);
    this->payloadLength = payloadLength;
}
```

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

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

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

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

```
class FooPacket_Base : public cPacket
{
  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 will find it the generated C++ header too, as a comment):

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

Register_Class(FooPacket);
```

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

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

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

    virtual void setPayloadLength(int newlength);
}

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

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

6.7.3 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 yet 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:

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

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

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

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

6.8 Using Standard Container Classes for Fields

One often wants to use standard container classes (STL) as fields, such as `std::vector`, `std::stack` or `std::map`. The following sections describe two ways this can be done:

1. via a typedef;
2. by defining the field as abstract, and customizing the generated class.

6.8.1 Typedefs

The basic idea is that if we create a typedef for the desired type, we can use it for fields just as any other type. Example:

```
cplusplus {{
#include <vector>
typedef std::vector<int> IntVector;
}}

class noncobject IntVector;

packet FooPacket {
    IntVector addresses;
};
```

The generated class will have the following methods:

```
virtual IntVector& getAddresses();
virtual const IntVector& getAddresses() const;
virtual void setAddresses(const IntVector& addresses);
```

Thus, the underlying `std::vector<int>` is exposed and you can directly manipulate it from C++ code, for example like this:

```
FooPacket *pk = new FooPacket();
pk->getAddresses().push_back(1);
pk->getAddresses().push_back(5);
pk->getAddresses().push_back(9);
// or:
IntVector& v = pk->getAddresses();
v.push_back(1);
v.push_back(5);
v.push_back(9);
```

It is easy. However, there are also some drawbacks:

1. The message compiler won't know that your field is actually a data structure, so the generated reflection code won't be able to look into it;
2. The fact that STL classes are directly exposed may be a mixed blessing; on one hand this makes it easier to manipulate its contents, but on the other hand it violates the encapsulation principle. Container classes work best when they are used as "nuts and bolts" for your C++ program, but they shouldn't really be used as public API.

6.8.2 Abstract Fields

This approach uses abstract fields. We exploit the fact that `std::vector` and `std::stack` are representations of *sequence*, which is the same abstraction as fields' variable-size array. That is, if you declare the field to be `abstract fieldname[]`, the message compiler will only generate pure virtual functions and you can implement the underlying data storage using standard container classes. You can also write additional C++ methods that delegate to the container object's `push_back()`, `push()`, `pop()`, etc. methods.

Consider the following message declaration:

```
packet FooPacket
{
    @customize(true);
    abstract int foo[]; // will use std::vector<int>
    abstract int bar[]; // will use std::stack<int>
}
```

If you compile the above code, in the generated C++ code you will only find abstract methods for `foo` and `bar`, but no underlying data members or method implementations. You can implement everything as you like. You can write the following C++ file then to implement `foo` and `bar` with `std::vector` and `std::stack` (some details omitted for brevity):

```
#include <vector>
#include <stack>
#include "FooPacket_m.h"

class FooPacket : public FooPacket_Base
{
  protected:
    std::vector<int> foo;
```

```
    std::stack<int> bar;

    // helper method
    void unsupported() {throw cRuntimeError("unsupported method called");}

  public:
    ...
    // foo methods
    virtual int getFoo(unsigned int k) {return foo[k];}
    virtual void setFoo(unsigned int k, int x) {foo[k]=x;}
    virtual void addFoo(int x) {foo.push_back(x);}
    virtual void setFooArraySize(unsigned int size) {foo.resize(size);}
    virtual unsigned int getFooArraySize() const {return foo.size();}

    // bar methods
    virtual int getBar(unsigned int k) {...}
    virtual void setBar(unsigned int k, int x) {unsupported();}
    virtual void barPush(int x) {bar.push(x);}
    virtual void barPop() {bar.pop();}
    virtual int barTop() {return bar.top();}
    virtual void setBarArraySize(unsigned int size) {unsupported();}
    virtual unsigned int getBarArraySize() const {return bar.size();}
};

Register_Class(FooPacket);
```

Some additional boilerplate code is needed so that the class conforms to conventions, and duplication and copying works properly:

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

Some additional notes:

1. `setFooArraySize()`, `setBarArraySize()` are redundant.
2. `getBar(int k)` cannot be implemented in a straightforward way (`std::stack` does not support accessing elements by index). It could still be implemented in a less efficient way using STL iterators, and efficiency does not seem to be major problem because only Tkenv is going to invoke this function.

3. `setBar(int k, int x)` could not be implemented, but this is not particularly a problem. The exception will materialize in a Tkenv error dialog when you try to change the field value.

6.9 Namespaces

It is possible to place the generated classes into a C++ namespace, and also to use types from other namespaces.

6.9.1 Declaring a Namespace

To place the generated types into a namespace, add a namespace declaration near the top of the message file:

```
namespace inet;
```

If you are fond of hierarchical (nested) namespaces, you can declare one with a straightforward syntax, using double colons in the namespace declaration. There is no need for multiple nested `namespace` declarations as in C++:

```
namespace org::omnetpp::inet::ieee80211;
```

The above code will be translated into nested namespaces in the C++ code:

```
namespace org { namespace omnetpp { namespace inet { namespace ieee80211 {
...
}}}}
```

Conceptually, the namespace extends from the place of the namespace declaration to the end of the message file. (A message file may contain only one namespace declaration.) In other words, it *does* matter whether you put something above the namespace declaration line or below it:

1. The contents of **`cplusplus`** blocks above the namespace declaration will be placed outside (i.e. above) the namespace block in the generated C++ header; blocks below the namespace declaration will placed inside the C++ namespace block.
2. Type announcements are interpreted differently depending on whether they occur above or below the namespace declaration (this will be detailed later).
3. Types defined with the message syntax are placed into the namespace of the message file; thus, definitions must always be *after* the namespace declaration. Type definitions above the namespace line will be rejected with an error message.

6.9.2 C++ Blocks and Namespace

As described above, the contents of a **`cplusplus`** block will be copied above or into the C++ namespace block in the generated header depending on whether it occurs above or below the namespace declaration in the message file.

The placement of **`cplusplus`** blocks relative to the namespace declaration is important because you don't want `#include` directives to be placed inside the C++ namespace block. That

would cause the declarations in the header file to be interpreted as being part of the namespace, which they are not. Includes should always be put into `cplusplus` blocks *above* the namespace declaration. This is so important that I repeat it:

> **IMPORTANT:** Includes should always be placed into a **`cplusplus`** block **above** the namespace declaration.

As for typedefs and other C++ code, you need to place them above or below the namespace declaration based on whether you want them to be in the C++ namespace or not.

6.9.3 Type Announcements and Namespace

The type announcement syntax allows one to specify the namespace of the type as well, so the following lines are syntactically correct:

```
packet foo::FooPacket;
packet nes::ted::name::space::BarPacket;
packet ::BazPacket;
```

Announced type names are interpreted in the following way:

1. If the type name contains a double colon (`::`), it is interpreted as being fully qualified with an absolute namespace.
2. If the name is just an identifier (no double colon), the interpretation depends on whether it is above or below the namespace declaration. If it is above, the name is interpreted as a global type; otherwise it is interpreted as part of the package file's namespace.

This also means that if you want to announce a global type, you either have to put the announcement above the namespace declaration, or prefix the type with "`::`" to declare that it is not part of a namespace.

When the announced types are used later (as field type, base class, etc.), they can be referred to just with their simple names (without namespace); or alternatively with their fully qualified names. When a message compiler encounters type name as field type or base class, it interprets the type name in the following way:

1. If the type name contains a double colon (`::`), it is interpreted as being fully qualified with an absolute namespace.
2. If the name is just an identifier (no double colon), and the message file's namespace contains that name, it is chosen; otherwise:
3. It is looked up among all announced types in all namespaces (including the global namespace), and there must be exactly one match. That is, if the same name exists in multiple namespaces, it may only be referenced with fully qualified name.

The following code illustrates the above rules:

```
cplusplus {{
// includes go above the namespace line
#include <vector>
#include "IPAddress.h"
```

```
}}

// the IPAddress type is in the global namespace
class noncobject IPAddress;

namespace foo;  // namespace begins with this line

// we could also have announced IPAddress here as "::IPAddress":
//class noncobject ::IPAddress;

cplusplus {{
// we want IPAddressVector to be part of the namespace
typedef std::vector<IPAddress> IPAddressVector;
}}

// type will be understood as foo::IPAddressVector
class noncobject IPAddressVector;

packet FooPacket {
    IPAddress source;
    IPAddressVector neighbors;
};
```

Another example that uses a `PacketData` class and a `NetworkPacket` type from a `net` namespace:

```
// NetworkPacket.msg
namespace net;
class PacketData { }
packet NetworkPacket { }
```

```
// FooPacket.msg
cplusplus {{
#include "NetworkPacket_m.h"
}}
class net::PacketData;
packet net::NetworkPacket;

namespace foo;

packet FooPacket extends NetworkPacket
{
    PacketData data;
}
```

6.10 Descriptor Classes

For each generated class and struct, the message compiler generates an associated descriptor class. The descriptor class carries "reflection" information about the new class, and makes it possible to inspect message contents in Tkenv.

The descriptor class encapsulates virtually all information that the original message definition contains, and exposes it via member functions. It has methods for enumerating fields (`getFieldCount()`, `getFieldName()`, `getFieldTypeString()`, etc.), for getting and setting a field's value in an instance of the class (`getFieldAsString()`, `setFieldAsString()`), for exploring the class hierarchy (`getBaseClassDescriptor()`, etc.), for accessing class and field properties, and for similar tasks. When you inspect a message or packet in the simulation, Tkenv can uses the associated descriptor class to extract and display the field values.

The `@descriptor` class property can be used to control the generation of the descriptor class. `@descriptor(readonly)` instructs the message compiler not to generate field setters for the descriptor, and `@descriptor(false)` instructs it not to generate a description class for the class at all.

It is also possible to use (or abuse) the message compiler for generating a descriptor class for an existing class. (This can be useful for making your class inspectable in Tkenv.) To do that, write a message definition for your existing class (for example, if it has `int getFoo()` and `setFoo(int)` methods, add an `int foo` field to the message definition), and mark it with `@existingClass(true)`. This will tell the message compiler that it should not generate an actual class (as it already exists), only a descriptor class.

6.11 Summary

This section summarizes the possibilities offered by message definitions.

Base functionality:

- generation of classes and plain C structs from concise descriptions
- default base classes: `cPacket` (with the **`packet`** keyword), `cMessage` (with the **`message`** keyword), or `cObject` (with the **`class`** keyword)

The following data types are supported for fields:

- primitive types: `bool`, `char`, `short`, `int`, `long`; `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`; `int8_t`, `int16_t`, `int32_t`, `int64_t`; `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`; `float`, `double`; `simtime_t`
- `string`, a dynamically allocated string, presented as `const char *`
- structs and classes, declared with the message syntax or in C++ code
- typedef'd names declared in C++ and announced to the message compiler
- fixed-size arrays of the above types
- variable-size 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 for struct/class fields)
- field initializers can be specified (except for struct/class fields)
- associating fields of integral types with enums

- inheritance
- namespaces
- customization of generated method names
- customization of the generated class via subclassing (*Generation Gap* pattern)
- abstract fields (for nonstandard storage and calculated fields)
- generation of descriptor objects that encapsulate reflection information

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();`

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

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

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

Chapter 7

The Simulation Library

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

- the message class `cMessage` (chapter 5)
- sending and receiving messages, scheduling and canceling events, terminating the module or the simulation (section 4.7)
- access to module gates and parameters via `cModule` member functions (sections 4.5 and 4.6)
- accessing other modules in the network (section 4.11)
- dynamic module creation (section 4.13)

This chapter discusses the rest of the simulation library:

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

7.1 Class Library Conventions

7.1.1 Base Class

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

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

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

7.1.2 Setting and Getting Attributes

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

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

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

7.1.3 getClassName()

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

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

7.1.4 Object Names

Gates, parameters and modules all have names in the NED files. At runtime, those names are stored in the corresponding C++ objects, and are available for the code with the `getName()` method. Other objects such as messages, queues, result collection objects, etc. may also

have names. [1] Object names are generally very useful when you are tracing, debugging or demonstrating a simulation model.

For example, you can get the name of a message object like this:

```
const char *name = msg->getName();
```

The `getName()` method will never return `NULL`, the absence of name string is always returned as the empty string (`""`).

By convention, the object name is the first argument to the constructor of every class, and it defaults to the empty string. To create an object with a name, pass the name string (a `const char *` pointer) as the first argument of the constructor:

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

You can also change the name of an existing object:

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

Both the constructor and `setName()` make an internal copy of the string, instead of just storing the pointer passed to them. [2]

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

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

7.1.5 Object Full Name and Full Path

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

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

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

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

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

[1] Object name is inherited from `cObject` (which defines `getName()`), and from its subclass `cNamedObject` (which defines `setName()` and actually stores the name string).

[2] In a simulation, there are usually many objects with the same name: modules, parameters, gates, etc. To conserve memory, several classes keep names in a shared, reference-counted *name pool* instead of making separate copies for each object. The runtime cost of looking up an existing string in the name pool and incrementing its reference count also compares favorably to the cost of allocation and copying.

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

7.1.6 Copying and Duplicating Objects

The `dup()` member function creates an exact copy of the object, duplicating contained objects also if necessary. This is especially useful in the case of message objects.

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

`dup()` delegates to the copy constructor. Classes also declare an assignment operator (`operator=()`) which can be used to copy contents of an object into another object of the same type. `dup()`, the copy constructor and the assignment operator all perform deep coping: objects contained in the copied object will also be duplicated if necessary.

`operator=()` differs from the other two in that it does *not* copy the object's name string, i.e. does not invoke `setName()`. The rationale is that the name string is often used for identifying the particular object instance, as opposed to being considered as part of its contents.

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

7.1.7 Iterators

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

For example:

```
cQueue queue;

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

7.1.8 Error Handling

When library objects detect an error condition, they throw a C++ exception. This exception is then caught by the simulation environment which pops up an error dialog or displays the error message.

At times it can be useful to be able stop the simulation at the place of the error (just before the exception is thrown) and use a C++ debugger to look at the stack trace and examine variables. Enabling the `debug-on-errors` ini file entry lets you do that – check it in section 10.2 .

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

An example:

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

7.2 Logging from Modules

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

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

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

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

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

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

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

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

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

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

`EV`'s implementation makes use of the fact that the « operator is bound more loosely than the conditional operator (`?:`).

7.3 Simulation Time Conversion

Simulation time is represented by the type `simtime_t` which is a typedef to a class that stores simulation time in a 64-bit integer, using decimal ïňĄxed-point representation. OMNeT++ provides utility functions, which convert `simtime_t` to a printable string (`"3s 130ms 230us"`) and vica versa.

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

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

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

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

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

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

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

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

7.4 Generating Random Numbers

Random numbers in simulation are never random. Rather, they are produced using deterministic algorithms. Algorithms take a *seed* value and perform some deterministic calculations on them to produce a "random" number and the next seed. Such algorithms and their implementations are called *random number generators* or RNGs, or sometimes pseudo random number generators or PRNGs to highlight their deterministic nature. [3]

Starting from the same seed, RNGs always produce the same sequence of random numbers. This is a useful property and of great importance, because it makes simulation runs repeatable.

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

7.4.1 Random Number Generators

Mersenne Twister

By default, OMNeT++ uses the Mersenne Twister RNG (MT) by M. Matsumoto and T. Nishimura [MN98]. MT has a period of $2^{19937} - 1$, and 623-dimensional equidistribution property is assured. MT is also very fast: as fast or faster than ANSI C's `rand()`.

The "Minimal Standard" RNG

OMNeT++ releases prior to 3.0 used a linear congruential generator (LCG) with a cycle length of $2^{31} - 2$, described in [Jai91], pp. 441-444,455. This RNG is still available and can be selected from `omnetpp.ini` (Chapter 11). This RNG is only suitable for small-scale simulation studies. As shown by Karl Entacher et al. in [EHW02], the cycle length of about 2^{31} is too small (on

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

todays fast computers it is easy to exhaust all random numbers), and the structure of the generated "random" points is too regular. The [Hel98] paper provides a broader overview of issues associated with RNGs used for simulation, and it is well worth reading. It also contains useful links and references on the topic.

The Akaroa RNG

When you execute simulations under Akaroa control (see section 11.5), you can also select Akaroa's RNG as the RNG underlying for the OMNeT++ random number functions. The Akaroa RNG also has to be selected from `omnetpp.ini` (section 10.5).

Other RNGs

OMNeT++ allows plugging in your own RNGs as well. This mechanism, based on the `cRNG` interface, is described in section 16.1. For example, one candidate to include could be L'Ecuyer's CMRG [LSCK02] which has a period of about 2^{191} and can provide a large number of *guaranteed* independent streams.

7.4.2 Random Number Streams, RNG Mapping

Simulation programs may consume random numbers from several streams, that is, from several independent RNG instances. For example, if a network simulation uses random numbers for generating packets and for simulating bit errors in the transmission, it might be a good idea to use different random streams for both. Since the seeds for each stream can be configured independently, this arrangement would allow you to perform several simulation runs with the same traffic but with bit errors occurring in different places. A simulation technique called *variance reduction* is also related to the use of different random number streams.

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

The number of random number streams as well as seeds for the individual streams can be configured in `omnetpp.ini` (section 10.5). For the "minimal standard RNG", the `seedtool` program can be used for selecting good seeds (section 10.5.5).

In OMNeT++, streams are identified with RNG numbers. The RNG numbers used in simple modules may be *arbitrarily mapped* to the actual random number streams (actual RNG instances) from `omnetpp.ini` (section 10.5). The mapping allows for great flexibility in RNG usage and random number streams configuration – even for simulation models which were not written with RNG awareness.

7.4.3 Accessing The RNGs

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

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

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

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

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

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

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

7.4.4 Random Variates

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

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

OMNeT++ has the following predefined distributions:

Function	**Description**
Continuous distributions	
`uniform(a, b, rng=0)`	uniform distribution in the range [a,b)
`exponential(mean, rng=0)`	exponential distribution with the given mean
`normal(mean, stddev, rng=0)`	normal distribution with the given mean and standard deviation
`truncnormal(mean, stddev, rng=0)`	normal distribution truncated to nonnegative values
`gamma_d(alpha, beta, rng=0)`	gamma distribution with parameters alpha>0, beta>0
`beta(alpha1, alpha2, rng=0)`	beta distribution with parameters alpha1>0, alpha2>0
`erlang_k(k, mean, rng=0)`	Erlang distribution with k>0 phases and the given mean
`chi_square(k, rng=0)`	chi-square distribution with k>0 degrees of freedom
`student_t(i, rng=0)`	student-t distribution with i>0 degrees of freedom
`cauchy(a, b, rng=0)`	Cauchy distribution with parameters a,b where b>0
`triang(a, b, c, rng=0)`	triangular distribution with parameters a<=b<=c, a!=c
`lognormal(m, s, rng=0)`	lognormal distribution with mean m and variance s>0
`weibull(a, b, rng=0)`	Weibull distribution with parameters a>0, b>0
`pareto_shifted(a, b, c, rng=0)`	generalized Pareto distribution with parameters a, b and shift c

Discrete distributions	
`intuniform(a, b, rng=0)`	uniform integer from a..b
`bernoulli(p, rng=0)`	result of a Bernoulli trial with probability 0<=p<=1 (1 with probability p and 0 with probability (1-p))
`binomial(n, p, rng=0)`	binomial distribution with parameters n>=0 and 0<=p<=1
`geometric(p, rng=0)`	geometric distribution with parameter 0<=p<=1
`negbinomial(n, p, rng=0)`	negative 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 non-negative 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 don't suffice, you can write your own functions. If you register your functions with the `Define_NED_Function()` macro, you can use them in NED files and ini files too.

7.4.5 Random Numbers from Histograms

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

7.5 Container Classes

7.5.1 Queue class: cQueue

Basic Usage

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

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++)
```

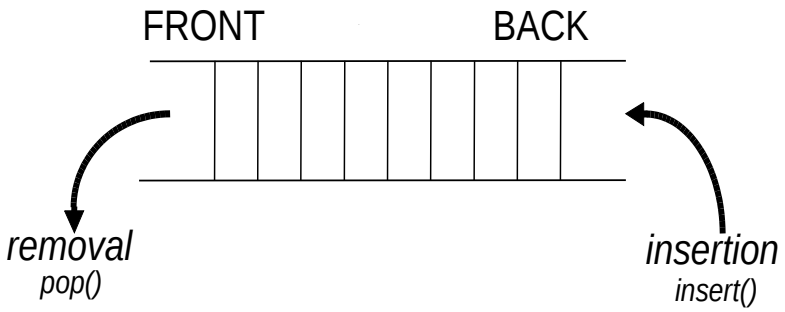


Figure 7.1: cQueue: insertion and removal

```
{
  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 is anything in the queue.

There are other functions dealing with insertion and removal. The `insertBefore()` and `insertAfter()` functions insert a new item exactly before or after a specified one, regardless of the ordering function.

The `front()` and `back()` functions return pointers to the objects at the front and back of the queue, without affecting queue contents.

The `pop()` function can be used to remove items from the tail of the queue, and the `remove()` function can be used to remove any item known by its pointer from the queue:

```
queue.remove(msg);
```

Priority Queue

By default, `cQueue` implements a FIFO, but it can also act as a priority queue, that is, it can keep the inserted objects ordered. 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 queue("queue", someCompareFunc);
```

If the queue object is set up as an ordered queue, the `insert()` function uses the ordering function: it searches the queue contents from the head until it reaches the position where the new item needs to be inserted, and inserts it there.

Iterators

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 argument 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 are at the end (or the beginning) of the queue.

An example:

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

7.5.2 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 it grows automatically when it becomes full. Internally, `cArray` is implemented with an array of pointers; when the array fills up, it is reallocated.

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

Creating an array:

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

Adding an object at the first free index:

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

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

```
cPar *p = new cMsgPar("par");
int index = array.addAt(5,p);
```

Finding an object in the array:

```
int index = array.find(p);
```

Getting a pointer to an object at a given index:

```
cPar *p = (cPar *) array[index];
```

You can also search the array or get a pointer to an object by the object's name:

```
int index = array.find("par");
Par *p = (cPar *) array["par"];
```

You can remove an object from the array by calling `remove()` with the object name, the index position or the object pointer:

```
array.remove("par");
array.remove(index);
array.remove( p );
```

The `remove()` function doesn't deallocate the object, but it returns the object pointer. If you also want to deallocate it, you can write:

```
delete array.remove( index );
```

Iteration

`cArray` has no iterator, but it is easy to loop through all the indices with an integer variable. The `size()` member function returns the largest index plus one.

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

7.6 Routing Support: cTopology

7.6.1 Overview

The `cTopology` class was designed primarily to support routing in 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 is no submodule nesting. Connections which span across compound module boundaries are also represented as one graph edge. Graph edges are directed, just as module gates are.

If you are writing a router or switch model, the `cTopology` graph can help you determine what nodes are available through which gate and also to find optimal routes. The `cTopology` object can calculate shortest paths between nodes for you.

The mapping between the graph (nodes, edges) and network model (modules, gates, connections) is preserved: you can easily find the corresponding module for a `cTopology` node and vica versa.

7.6.2 Basic Usage

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

- by module type
- by a parameter's presence and its value
- with a user-supplied boolean function

First, you can specify which node types you want to include. The following code extracts all modules of type `Router` or `Host`. (`Router` and `Host` can be either simple or compound module types.)

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

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

A dynamically assembled list of module types can be passed as a `NULL`-terminated array of `const char*` pointers, or in an STL string vector `std::vector<std::string>`. An example for the former:

```
cTopology topo;
const char *typeNames[3];
typeNames[0] = "Router";
typeNames[1] = "Host";
typeNames[2] = NULL;
topo.extractByModuleType(typeNames);
```

Second, you can extract all modules which have a certain parameter:

```
topo.extractByParameter( "ipAddress" );
```

You can also specify that the parameter must have a certain value for the module to be included in the graph:

```
cMsgPar yes = "yes";
topo.extractByParameter( "includeInTopo", &yes );
```

The third form allows you to pass a function which can determine for each module whether it should or should not be included. You can have `cTopology` pass supplemental data to the function through a `void*` pointer. An example which selects all top-level modules (and does not use the `void*` pointer):

```
int selectFunction(cModule *mod, void *)
{
  return mod->getParentModule() == simulation.getSystemModule();
}

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

A `cTopology` object uses two types: `cTopology::Node` for nodes and `cTopology::Link` for edges. (`sTopoLinkIn` and `cTopology::LinkOut` are 'aliases' for `cTopology::Link`; we'll talk about them later.)

Once you have the topology extracted, you can start exploring it. Consider the following code (we'll explain it shortly):

```
for (int i=0; i<topo.getNumNodes(); i++)
{
  cTopology::Node *node = topo.getNode(i);
  ev << "Node i=" << i << " is " << node->getModule()->getFullPath() << endl;
  ev << " It has " << node->getNumOutLinks() << " conns to other nodes\n";
  ev << " and " << node->getNumInLinks() << " conns from other nodes\n";

  ev << " Connections to other modules are:\n";
  for (int j=0; j<node->getNumOutLinks(); j++)
  {
    cTopology::Node *neighbour = node->getLinkOut(j)->getRemoteNode();
    cGate *gate = node->getLinkOut(j)->getLocalGate();
    ev << " " << neighbour->getModule()->getFullPath()
       << " through gate " << gate->getFullName() << endl;
  }
}
```

The `getNumNodes()` member function (1st line) returns the number of nodes in the graph, and getNode(i) returns a pointer to the *i*th node, an `cTopology::Node` structure.

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

```
cTopology::Node *node = topo.getNodeFor( module );
cModule *module = node->getModule();
```

The `getNodeFor()` member function returns a pointer to the graph node for a given module. (If the module is not in the graph, it returns `NULL`). `getNodeFor()` uses binary search within the `cTopology` object so it is relatively fast.

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

By calling member functions of the graph edge object, you can determine the modules and gates involved. The `getRemoteNode()` function returns the other end of the connection, and `getLocalGate()`, `getRemoteGate()`, `getLocalGateId()` and `getRemoteGateId()` return the gate pointers and ids of the gates involved. (Actually, the implementation is a bit tricky here: the same graph edge object `cTopology::Link` is returned either as `cTopology::LinkIn` or as `cTopology::LinkOut` so that "remote" and "local" can be correctly interpreted for edges of both directions.)

7.6.3 Shortest Paths

The real power of `cTopology` is in finding shortest paths in the network to support optimal routing. `cTopology` finds shortest paths from *all* nodes *to* a target node. The algorithm is computationally inexpensive. In the simplest case, all edges are assumed to have the same weight.

A real-life example assumes we have the target module pointer; finding the shortest path to the target looks like this:

```
cModule *targetmodulep =...;
cTopology::Node *targetnode = topo.getNodeFor( targetmodulep );
topo.calculateUnweightedSingleShortestPathsTo( targetnode );
```

This performs the Dijkstra algorithm and stores the result in the `cTopology` object. The result can then be extracted using `cTopology` and `cTopology::Node` methods. Naturally, each call to `calculateUnweightedSingleShortestPathsTo()` overwrites the results of the previous call.

Walking along the path from our module to the target node:

```
cTopology::Node *node = topo.getNodeFor( this );

if (node == NULL)
{
  ev < "We (" << getFullPath() << ") are not included in the topology.\n";
}
else if (node->getNumPaths()==0)
{
  ev << "No path to destination.\n";
}
else
{
  while (node != topo.getTargetNode())
  {
    ev << "We are in " << node->getModule()->getFullPath() << endl;
    ev << node->getDistanceToTarget() << " hops to go\n";
    ev << "There are " << node->getNumPaths()
       << " equally good directions, taking the first one\n";
    cTopology::LinkOut *path = node->getPath(0);
    ev << "Taking gate " << path->getLocalGate()->getFullName()
       << " we arrive in " << path->getRemoteNode()->getModule()->getFullPath()
       << " on its gate " << path->getRemoteGate()->getFullName() << endl;
    node = path->getRemoteNode();
  }
}
```

The purpose of the `getDistanceToTarget()` member function of a node is self-explanatory. In the unweighted case, it returns the number of hops. The `getNumPaths()` member function returns the number of edges which are part of a shortest path, and `path(i)` returns the *i*th edge of them as `cTopology::LinkOut`. If the shortest paths were created by the `...SingleShortestPaths()` function, `getNumPaths()` will always return 1 (or 0 if the target is not reachable), that is, only one of the several possible shortest paths are found. The `...MultiShortestPathsTo()` functions find all paths, at increased run-time cost. The `cTopology`'s `getTargetNode()` function returns the target node of the last shortest path search.

You can enable/disable nodes or edges in the graph. This is done by calling their `enable()` or `disable()` member functions. Disabled nodes or edges are ignored by the shortest paths calculation algorithm. The `isEnabled()` member function returns the state of a node or edge in the topology graph.

One usage of `disable()` is when you want to determine in how many hops the target node can be reached from our node *through a particular output gate*. To compute this, you compute the shortest paths to the target *from the neighbor node* while disabling the current node to prevent the shortest paths from going through it:

```
cTopology::Node *thisnode = topo.getNodeFor( this );
thisnode->disable();
topo.calculateUnweightedSingleShortestPathsTo( targetnode );
```

```
thisnode->enable();

for (int j=0; j<thisnode->getNumOutLinks(); j++)
{
  cTopology::LinkOut *link = thisnode->getLinkOut(i);
  ev << "Through gate " << link->getLocalGate()->getFullName() << " : "
     << 1 + link->getRemoteNode()->getDistanceToTarget() << " hops" << endl;
}
```

In the future, other shortest path algorithms will also be implemented:

```
unweightedMultiShortestPathsTo(cTopology::Node *target);
weightedSingleShortestPathsTo(cTopology::Node *target);
weightedMultiShortestPathsTo(cTopology::Node *target);
```

7.7 Pattern Matching

Since version 4.3, OMNeT++ contains two utility classes for pattern matching, `cPatternMatcher` and `cMatchExpression`.

`cPatternMatcher` is a glob-style pattern matching class, adopted to special OMNeT++ requirements. It recognizes wildcards, character ranges and numeric ranges, and supports options such as case sensitive and whole string matching. `cMatchExpression` builds on top of `cPatternMatcher` and extends it in two ways: first, it lets you combine patterns with AND, OR, NOT into boolean expressions, and second, it applies the pattern expressions to *objects* instead of text. These classes are especially useful for making model-specific configuration files more concise or more powerful by introducing patterns.

7.7.1 cPatternMatcher

`cPatternMatcher` holds a pattern string and several option flags, and has a boolean `matches()` function that lets you check whether the string passed as argument matches the pattern with the given flags. The pattern and the flags can be set via the constructor or by calling the `setPattern()` member function.

The pattern syntax is a variation on Unix *glob*-style patterns. The most apparent differences to globbing rules are the distinction between `*` and `**`, and that character ranges should be written with curly braces instead of square brackets; that is, *any-letter* is expressed as `{a-zA-Z}` and not as `[a-zA-Z]`, because square brackets are reserved for the notation of module vector indices.

The following option flags are supported:

- *dottedpath*: controls whether some wildcards (`?`, `*`) will match dots
- *fullstring*: controls whether to do full string or substring match.
- *casesensitive*: whether matching is case sensitive or case insensitive

Patterns may contain the following elements:

- *question mark*, `?` : matches any character (except dot if *dottedpath*=true)

- *asterisk*, `*` : matches zero or more characters (except dots if *dottedpath*=true)
- *double asterisk*, `**` : matches zero or more characters, including dots
- *set*, e.g. `{a-zA-Z}` : matches any character that is contained in the set
- *negated set*, e.g. `{^a-z}`: matches any character that is NOT contained in the set
- *numeric range*, e.g. `{38..150}` : matches any number (i.e. sequence of digits) in the given range
- *numeric index range*, e.g. `[38..150]` : matches any number in square brackets in the given range
- *backslash*, `\` : takes away the special meaning of the subsequent character

NOTE: The *dottedpath* option was introduced to make matching OMNeT++ module paths more powerful. When it is off (*dottedpath*=false), there is no difference between `*` and `**`, they both match any character sequence. However, when matching OMNeT++ module paths or other strings where dot is a separator character, it is useful to turn on the dottedpath mode (*dottedpath*=true). In that mode, `*`, not being able to cross a dot, can match only a single path component (or part of it), and `**` can match multiple path components.

Sets and negated sets can contain several character ranges and also enumeration of characters, for example `{_a-zA-Z0-9}` or `{xyzc-f}`. To include a hyphen in the set, place it at a position where it cannot be interpreted as character range, for example `{a-z-}` or `{-a-z}`. If you want to include a close brace in the set, it must be the first character: `{}a-z}`, or for a negated set: `{^}a-z}`. A backslash is always taken as literal backslash (and NOT as escape character) within set definitions. When doing case-insensitive match, avoid ranges that include both alpha and non-alpha characters, because they might cause funny results.

For numeric ranges and numeric index ranges, ranges are inclusive, and both the start and the end of the range are optional; that is, `{10..}`, `{..99}` and `{..}` are all valid numeric ranges (the last one matches any number). Only nonnegative integers can be matched. Caveat: `{17..19}` will match `"a17"`, `"117"` and also `"963217"`!

The `cPatternMatcher` constructor and the `setPattern()` member function have similar signatures:

```
cPatternMatcher(const char *pattern, bool dottedpath, bool fullstring,
                bool casesensitive);
void setPattern(const char *pattern, bool dottedpath, bool fullstring,
                bool casesensitive);
```

The matcher function:

```
bool matches(const char *text);
```

There are also some more utility functions for printing the pattern, determining whether a pattern contains wildcards, etc.

Example:

```
cPatternMatcher matcher("**.host[*]", true, true, true);
ev << matcher.matches("Net.host[0]") << endl;  // -> true
ev << matcher.matches("Net.area1.host[0]") << endl;  // -> true
ev << matcher.matches("Net.host") << endl;  // -> false
ev << matcher.matches("Net.host[0].tcp") << endl;  // -> false
```

7.7.2 cMatchExpression

The `cMatchExpression` class builds on top of `cPatternMatcher`, and lets you determine whether an *object* matches a given pattern expression.

A pattern expression consists of elements in the *fieldname(pattern)* syntax; they check whether the string representation of the given field of the object matches the pattern. For example, `srcAddr(192.168.0.*)` will match if the *srcAddr* field of the object starts with `192.168.0.`. A naked pattern (without field name and parens) is also accepted, and it will be matched against the *default field* of the object, which will usually be its name.

These elements can be combined with the AND, OR, NOT operators, accepted in both lowercase and uppercase. AND has higher precedence than OR, but parentheses can be used to change the evaluation order.

Pattern examples:

- `"node*"`
- `"node* or host*"`
- `"packet-* and className(PPPFrame)"`
- `"className(TCPSegment) and byteLength({4096..})"`
- `"className(TCPSegment) and (SYN or DATA-*) and not kind({0..2})"`

The `cMatchExpression` class has a constructor and `setPattern()` method similar to those of `cPatternMatcher`:

```
cMatchExpression(const char *pattern, bool dottedpath, bool fullstring,
                 bool casesensitive);
void setPattern(const char *pattern, bool dottedpath, bool fullstring,
                 bool casesensitive);
```

However, the matcher function takes a `cMatchExpression::Matchable` instead of string:

```
bool matches(const Matchable *object);
```

This means that objects to be matched must either be subclassed from `cMatchExpression::Matchable`, or be wrapped into some adapter class that does. `cMatchExpression::Matchable` is a small abstract class with only a few pure virtual functions:

```
/**
 * Objects to be matched must implement this interface
 */
class SIM_API Matchable
{
  public:
    /**
     * Return the default string to match. The returned pointer will not be
     * cached by the caller, so it is OK to return a pointer to a static buffer.
     */
    virtual const char *getAsString() const = 0;

    /**
     * Return the string value of the given attribute, or NULL if the object
```

```
     * doesn't have an attribute with that name. The returned pointer will not
     * be cached by the caller, so it is OK to return a pointer to a static buffer.
     */
    virtual const char *getAsString(const char *attribute) const = 0;

    /**
     * Virtual destructor.
     */
    virtual ~Matchable() {}
};
```

To be able to match instances of an existing class that is not already a `Matchable`, you need to write an adapter class. An adapter class that we can look at as an example is `cMatchableString`. `cMatchableString` makes it possible to match strings with a `cMatchExpression`, and is part of OMNeT++:

```
/**
 * Wrapper to make a string matchable with cMatchExpression.
 */
class cMatchableString : public cMatchExpression::Matchable
{
  private:
    std::string str;
  public:
    cMatchableString(const char *s) {str = s;}
    virtual const char *getAsString() const {return str.c_str();}
    virtual const char *getAsString(const char *name) const {return NULL;}
};
```

An example:

```
cMatchExpression expr("foo* or bar*", true, true, true);
cMatchableString str1("this is a foo");
cMatchableString str2("something else");
ev << expr.matches(&str1) << endl; // -> true
ev << expr.matches(&str2) << endl; // -> false
```

Or, by using temporaries:

```
ev << expr.matches(&cMatchableString("this is a foo")) << endl; // -> true
ev << expr.matches(&cMatchableString("something else")) << endl; // -> false
```

7.8 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 the count, mean, standard deviation, minimum and maximum value etc of the observations.

- cWeightedStdDev is similar to cStdDev, but accepts weighted observations. cWeightedStdDev can be used for example to calculate time average. It is the only weighted statistics class.
- cLongHistogram and cDoubleHistogram are descendants of cStdDev and also keep an approximation of the distribution of the observations using equidistant (equal-sized) cell histograms.
- cVarHistogram implements a histogram where cells do not need to be the same size. You can manually add the cell (bin) boundaries, or alternatively, automatically have a partitioning created where each bin has the same number of observations (or as close to that as possible).
- cPSquare is a class that uses the P^2 algorithm described in [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 from the object: getCount(), getMin(), getMax(), getMean(), getStddev(), getVariance(), getSum(), getSqrSum() with the obvious meanings. An example usage for cStdDev:

```
cStdDev stat("stat");

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

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

7.8.2 Distribution Estimation

Initialization and Usage

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

The following member functions exist for setting up the range and to specify how many observations should be used for automatically determining the range (these methods are part of cDensityEstBase):

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

```
setNumFirstVals(numFirstvals);
```

The following example creates a histogram with 20 cells and automatic range estimation:

```
cDoubleHistogram histogram("histogram", 20);
histogram.setRangeAuto(100,1.5);
```

Here, 20 is the number of cells (not including the underflow/overflow cells, see later), and 100 is the number of observations to be collected before setting up the cells. 1.5 is the range extension factor. It means that the actual range of the initial observations will be expanded 1.5 times and this expanded range will be used to lay out the cells. This method increases the chance that further observations fall in one of the cells and not outside the histogram range.

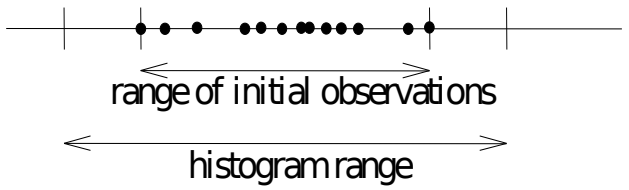


Figure 7.2: Setting up a histogram's range

The `isTransformed()` function returns *true* when the cells have already been set up. You can force range estimation and setting up the cells by calling the `transform()` function.

The observations that fall outside the histogram range will be counted as underflows and overflows. The number of underflows and overflows are returned by the `getUnderflowCell()` and `getOverflowCell()` member functions.

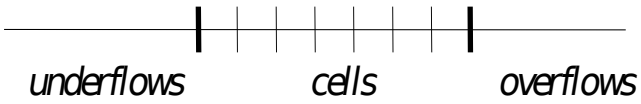


Figure 7.3: Histogram structure after setting up the cells

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 in each cell. `getNumCells()` returns the number of cells, `getBasepoint(int k)` returns the *k*th base point, `getCellValue(int k)` returns the number of observations in cell *k*, and `getCellPDF(int k)` returns the PDF value in the cell (i.e. between `getBasepoint(k)` and `getBasepoint(k+1)`). The `getCellInfo(k)` method returns multiple data (cell bounds,

counter, relative frequency) packed together in a struct. These functions work for all histogram types, plus `cPSquare` and `cKSplit`.

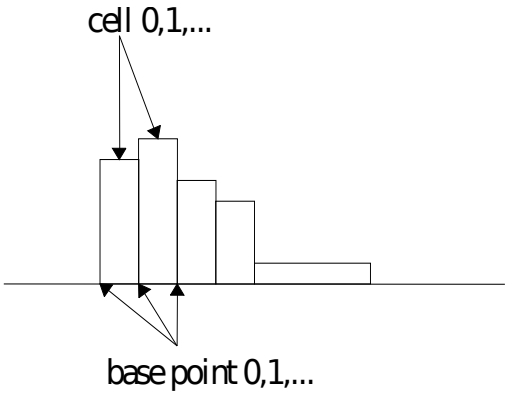


Figure 7.4: base points and cells

An example:

```
long n = histogram.getCount();
for (int i=0; i<histogram.getNumCells(); i++)
{
  double cellWidth = histogram.getBasepoint(i+1)-histogram.getBasepoint(i);
  int count = histogram.getCellValue(i);
  double pdf = histogram.getCellPDF(i);
  //...
}
```

The `getPDF(x)` and `getCDF(x)` member functions return the value of the Probability Density Function and the Cumulated Density Function at a given *x*, respectively.

Random Number Generation from Distributions

The `random()` member function generates random numbers from the distribution stored by the object:

```
double rnd = histogram.random();
```

`cStdDev` assumes normal distribution.

The `cPar` object stores the pointer to the histogram (or P^2 object), and whenever it is asked for the value, calls the histogram object's `random()` function:

```
double rnd = (double)rndPar; // random number from the cPSquare
```

Storing and Loading Distributions

The statistic classes have `loadFromFile()` member functions that read the histogram data from a text file. If you need a custom distribution that cannot be written (or it is inefficient) as a C function, you can describe it in histogram form stored in a text file, and use a histogram object with `loadFromFile()`.

You can also use `saveToFile()` that writes out the distribution collected by the histogram object:

```
FILE *f = fopen("histogram.dat","w");
histogram.saveToFile(f); // save the distribution
fclose(f);

cDoubleHistogram hist2("Hist-from-file");
FILE *f2 = fopen("histogram.dat","r");
hist2.loadFromFile(f2); // load stored distribution
fclose(f2);
```

Histogram with Custom Cells

The `cVarHistogram` class can be used to create histograms with arbitrary (non-equidistant) cells. It can operate in two modes:

- *manual*, where you specify cell boundaries explicitly before starting collecting
- *automatic*, where `transform()` will set up the cells after collecting a certain number of initial observations. The cells will be set up so that as far as possible, an equal number of observations fall into each cell (equi-probable cells).

Modes are selected with a *transform-type* parameter:

- `HIST_TR_NO_TRANSFORM`: no transformation; uses bin boundaries previously defined by `addBinBound()`
- `HIST_TR_AUTO_EPC_DBL`: automatically creates equiprobable cells
- `HIST_TR_AUTO_EPC_INT`: like the above, but for integers

Creating an object:

```
cVarHistogram(const char *s=NULL,
              int numcells=11,
              int transformtype=HIST_TR_AUTO_EPC_DBL);
```

Manually adding a cell boundary:

```
void addBinBound(double x);
```

Rangemin and rangemax is chosen after collecting the `numFirstVals` initial observations. One cannot add cell boundaries when the histogram has already been transformed.

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. Experience has shown that $k = 2$ works best.

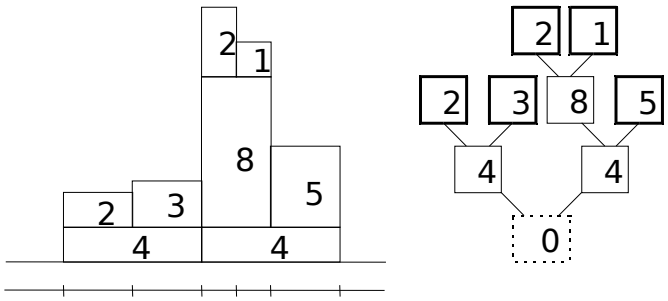


Figure 7.5: Illustration of the k-split algorithm, $k = 2$. The numbers in boxes represent the observation count values

For density estimation, the total number of observations that fell into each cell of the partition has to be determined. For this purpose, mother observations in each internal node of the tree must be distributed among its child cells and propagated up to the leaves.

Let $n_{...,i}$ be the (mother) observation count for a cell, $s_{...,i}$ be the total observation count in a cell $n_{...,i}$ plus the observation counts in all its sub-, sub-sub-, etc. cells), and $m_{...,i}$ the mother observations propagated to the cell. We are interested in the $\tilde{n}_{...,i} = n_{...,i} + m_{...,i}$ estimated amount of observations in the tree nodes, especially in the leaves. In other words, if we have $\tilde{n}_{...,i}$ estimated observation amount in a cell, how to divide it to obtain $m_{...,i,1}, m_{...,i,2} \cdots m_{...,i,k}$ that can be propagated to child cells. Naturally, $m_{...,i,1} + m_{...,i,2} + \cdots + m_{...,i,k} = \tilde{n}_{...,i}$.

Two natural distribution methods are even distribution (when $m_{...,i,1} = m_{...,i,2} = \cdots = m_{...,i,k}$) and proportional distribution (when $m_{...,i,1} : m_{...,i,2} : \cdots : m_{...,i,k} = s_{...,i,1} : s_{...,i,2} : \cdots : s_{...,i,k}$). Even distribution is optimal when the $s_{...,i,j}$ values are very small, and proportional distribution is good when the $s_{...,i,j}$ values are large compared to $m_{...,i,j}$. In practice, a linear combination of them seems appropriate, where $\lambda = 0$ means even and $\lambda = 1$ means proportional distribution:

$m_{...,i,j} = (1 - \lambda)\tilde{n}_{...,i}/k + \lambda\tilde{n}_{...,i}s_{...,i,j}/s_{...,i}$ where $\lambda \in [0, 1]$

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

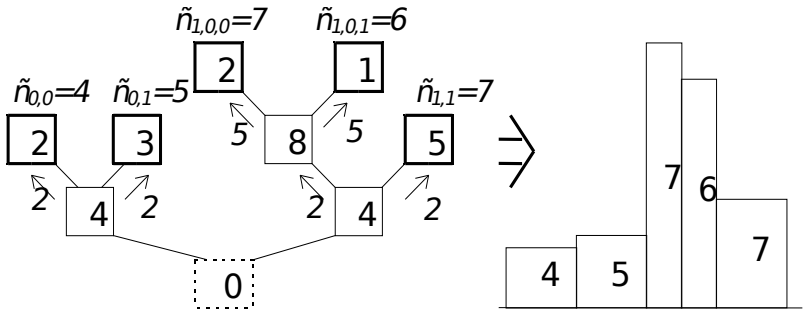


Figure 7.6: Density estimation from the k-split cell tree. We assume $\lambda = 0$, i.e. we distribute mother observations evenly.

set up the initial histogram range. Because of the range extension and cell split capabilities, the algorithm is not very sensitive to the choice of the initial range, so very few observations are sufficient for range estimation (say $N_{pre} = 10$). Thus we can regard *k*-split as an on-line method.

K-split can also be used in semi-online mode, when the algorithm is only used to create an optimal partition from a larger number of N_{pre} observations. When the partition has been created, the observation counts are cleared and the N_{pre} observations are fed into *k*-split once again. This way all mother (non-leaf) observation counts will be zero and the cell division error is eliminated. It has been shown that the partition created by *k*-split can be better than both the equi-distant and the equal-frequency partition.

OMNeT++ contains an implementation of the *k*-split algorithm, the `cKSplit` class.

The cKSplit Class

The `cKSplit` class is an implementation of the *k-split* method. Member functions:

```
void setCritFunc(KSplitCritFunc critfunc, double *critdata);
void setDivFunc(KSplitDivFunc divfunc, double *divdata);
void rangeExtension( bool enabled );
```

```
int getTreeDepth();
int getTreeDepth(cKSplit::Grid& grid);
```

```
double getRealCellValue(cKSplit::Grid& grid, int cell);
void printGrids();
```

```
cKSplit::Grid& getGrid(int k);
cKSplit::Grid& getRootGrid();
```

```
struct cKSplit::Grid
{
  int parent;   // index of parent grid
  int reldepth; // depth = (reldepth - rootgrid's reldepth)
  long total;   // sum of cells & all subgrids (includes "mother")
  int mother;   // observations "inherited" from mother cell
  int cells[K]; // cell values
};
```

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 has been collected to compute a 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 perform the specific algorithms on the data fed into the result object and determine if the transient period is over or the result accuracy has been reached.

The base classes for classes implementing specific transient detection and result accuracy detection algorithms are:

- `cTransientDetection`: base class for transient detection
- `cAccuracyDetection`: base class for result accuracy detection

Basic Usage

Attaching detection objects to a `cStatistic` and getting pointers to the attached objects:

```
addTransientDetection(cTransientDetection *object);
addAccuracyDetection(cAccuracyDetection *object);
cTransientDetection *getTransientDetectionObject();
cAccuracyDetection *getAccuracyDetectionObject();
```

Detecting the end of the period:

- polling the `detect()` function of the object
- installing a post-detect function

Transient Detection

Currently one transient detection algorithm is implemented, i.e. there is 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 is 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()` method is used to output a value (or a value pair) with a timestamp. The object name will serve as the name of the output vector.

The vector name can be passed in the constructor,

```
cOutVector responseTimeVec("response time");
```

but in the usual arrangement you'd make the `cOutVector` a member of the module class and set the name in `initialize()`. You'd record values from `handleMessage()` or from a function called from `handleMessage()`.

The following example is a `Sink` module which records the lifetime of every message that arrives to it.

```
class Sink : public cSimpleModule
{
  protected:
    cOutVector endToEndDelayVec;

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(Sink);

void Sink::initialize()
{
    endToEndDelayVec.setName("End-to-End Delay");
}

void Sink::handleMessage(cMessage *msg)
{
    simtime_t eed = simTime() - msg->getCreationTime();
    endToEndDelayVec.record(eed);
    delete msg;
}
```

There is also a `recordWithTimestamp()` method, to make it possible to record values into output vectors with a timestamp other than `simTime()`. Increasing timestamp order is still enforced though.

All `cOutVector` objects write to a single *output vector file* that has a file extension `.vec`. [4] The format and processing of output vector files is described in section 12.

You can configure output vectors from `omnetpp.ini`: you can disable individual vectors, or limit recording to certain simulation time intervals (section 12.2.5).

If the output vector object is disabled or the simulation time is outside the specified interval, `record()` doesn't write anything to the output file. However, if you have a Tkenv inspector

[4] A `.vci` file is also created, but it is just an index for the `.vec` file and does not contain any new information. The IDE re-creates the `.vci` file if it gets lost.

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 output scalars

- to record summary data at the end of the simulation run
- to do several runs with different parameter settings/random seed and determine the dependence of some measures on the parameter settings. For example, multiple runs and output scalars are the way to produce *Throughput vs. Offered Load* plots.

Output scalars are recorded with the `record()` method of `cSimpleModule`, and you will usually want to insert this code into the `finish()` function. An example:

```
void Transmitter::finish()
{
    double avgThroughput = totalBits / simTime();
    recordScalar("Average throughput", avgThroughput);
}
```

You can record whole statistic objects by calling their `record()` methods, declared as part of `cStatistic`. In the following example we create a `Sink` module which calculates the mean, standard deviation, minimum and maximum values of a variable, and records them at the end of the simulation.

```
class Sink : public cSimpleModule
{
  protected:
    cStdDev eedStats;

    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
};

Define_Module(Sink);

void Sink::initialize()
{
    eedStats.setName("End-to-End Delay");
}

void Sink::handleMessage(cMessage *msg)
{
    simtime_t eed = simTime() - msg->getCreationTime();
    eedStats.collect(eed);
    delete msg;
}
```

```
void Sink::finish()
{
    recordScalar("Simulation duration", simTime());
    eedStats.record();
}
```

The above calls record the data into an *output scalar file*, a line-oriented text file that has the file extension `.sca`. The format and processing of output vector files is described in chapter 12.

7.10 Watches and Snapshots

7.10.1 Basic Watches

Unfortunately, variables of type `int`, `long`, `double` do not show up by default in Tkenv; neither do STL classes (`std::string`, `std::vector`, etc.) or your own structs and classes. This is because the simulation kernel, being a library, knows nothing about types and variables in your source code.

OMNeT++ provides `WATCH()` and a set of other macros to allow variables to be inspectable in Tkenv and to be output into the snapshot file. `WATCH()` macros are usually placed into `initialize()` (to watch instance variables) or to the top of the `activity()` function (to watch its local variables); the point being that they should only be executed once.

```
long packetsSent;
double idleTime;

WATCH(packetsSent);
WATCH(idleTime);
```

Of course, members of classes and structs can also be watched:

```
WATCH(config.maxRetries);
```

When you open an inspector for the simple module in Tkenv and click the Objects/Watches tab in it, you will see your watched variables and their values there. Tkenv also lets you change the value of a watched variable.

The `WATCH()` macro can be used with any type that has a stream output operator (`operator«`) defined. By default, this includes all primitive types and `std::string`, but since you can write `operator«` for your classes/structs and basically any type, `WATCH()` can be used with anything. The only limitation is that since the output should more or less fit on single line, the amount of information that can be conveniently displayed is limited.

An example stream output operator:

```
std::ostream& operator<<(std::ostream& os, const ClientInfo& cli)
{
    os << "addr=" << cli.clientAddr << "  port=" << cli.clientPort; // no endl!
    return os;
}
```

And the `WATCH()` line:

```
WATCH(currentClientInfo);
```

7.10.2 Read-write Watches

Watches for primitive types and `std::string` allow for changing the value from the GUI as well, but for other types you need to explicitly add support for that. What you need to do is define a stream input operator (`operator»`) and use the `WATCH_RW()` macro instead of `WATCH()`.

The stream input operator:

```
std::ostream& operator>>(std::istream& is, ClientInfo& cli)
{
    // read a line from "is" and parse its contents into "cli"
    return is;
}
```

And the `WATCH_RW()` line:

```
WATCH_RW(currentClientInfo);
```

7.10.3 Structured Watches

`WATCH()` and `WATCH_RW()` are basic watches; they allow one line of (unstructured) text to be displayed. However, if you have a data structure generated from message definitions (see Chapter 5), then there is a better approach. The message compiler automatically generates meta-information describing individual fields of the class or struct, which makes it possible to display the contents on field level.

The `WATCH` macros to be used for this purpose are `WATCH_OBJ()` and `WATCH_PTR()`. Both expect the object to be subclassed from `cObject`; `WATCH_OBJ()` expects a reference to such class, and `WATCH_PTR()` expects a pointer variable.

```
ExtensionHeader hdr;
ExtensionHeader *hdrPtr;
...
WATCH_OBJ(hdr);
WATCH_PTR(hdrPtr);
```

CAUTION: With `WATCH_PTR()`, the pointer variable must point to a valid object or be `NULL` at all times, otherwise the GUI may crash while trying to display the object. This practically means that the pointer should be initialized to `NULL` even if not used, and should be set to `NULL` when the object to which it points is deleted.

```
delete watchedPtr;
watchedPtr = NULL;  // set to NULL when object gets deleted
```

7.10.4 STL Watches

The standard C++ container classes (`vector`, `map`, `set`, etc) also have structured watches, available via the following macros:

`WATCH_VECTOR()`, `WATCH_PTRVECTOR()`, `WATCH_LIST()`, `WATCH_PTRLIST()`, `WATCH_SET()`, `WATCH_PTRSET()` `WATCH_MAP()`, `WATCH_PTRMAP()`.

The `PTR`-less versions expect the data items ("T") to have stream output operators (`operator«`), because that is how they will display them. The `PTR` versions assume that data items are

pointers to some type which has `operator «`. `WATCH_PTRMAP()` assumes that only the value type ("second") is a pointer, the key type ("first") is not. (If you happen to use pointers as key, then define `operator «` for the pointer type itself.)

Examples:

```
std::vector<int> intvec;
WATCH_VECTOR(intvec);

std::map<std::string,Command*> commandMap;
WATCH_PTRMAP(commandMap);
```

7.10.5 Snapshots

The `snapshot()` function outputs textual information about all or selected objects of the simulation (including the objects created in module functions by the user) into the snapshot file.

```
bool snapshot(cObject *obj = &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`. The 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 snapshot file (some abbreviations have been applied):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<snapshot object="simulation" label="Long queue" simtime="9.038229311343"
network="FifoNet">
  <object class="cSimulation" fullpath="simulation">
    <info></info>
    <object class="cModule" fullpath="FifoNet">
      <info>id=1</info>
      <object class="fifo::Source" fullpath="FifoNet.gen">
        <info>id=2</info>
        <object class="cPar" fullpath="FifoNet.gen.sendIaTime">
          <info>exponential(0.01s)</info>
          <detailedinfo>double sendIaTime = exponential(0.01s) @unit(s)
          </detailedinfo>
        </object>
        <object class="cGate" fullpath="FifoNet.gen.out">
          <info>--&gt; fifo.in</info>
```

```
        </object>
      </object>
      <object class="fifo::Fifo" fullpath="FifoNet.fifo">
        <info>id=3</info>
        <object class="cPar" fullpath="FifoNet.fifo.serviceTime">
          <info>0.01</info>
          <detailedinfo>double serviceTime = 0.01@unit(s)</detailedinfo>
        </object>
        <object class="cGate" fullpath="FifoNet.fifo.in">
          <info>&lt;-- gen.out</info>
        </object>
        <object class="cGate" fullpath="FifoNet.fifo.out">
          <info>--&gt; sink.in</info>
        </object>
        <object class="cQueue" fullpath="FifoNet.fifo.queue">
          <info>length=13</info>
          <object class="cMessage" fullpath="FifoNet.fifo.queue.job">
            <info>src=FifoNet.gen (id=2)  dest=FifoNet.fifo (id=3)</info>
          </object>
          <object class="cMessage" fullpath="FifoNet.fifo.queue.job">
            <info>src=FifoNet.gen (id=2)  dest=FifoNet.fifo (id=3)</info>
          </object>
        </object>
      <object class="fifo::Sink" fullpath="FifoNet.sink">
        <info>id=4</info>
        <object class="cGate" fullpath="FifoNet.sink.in">
          <info>&lt;-- fifo.out</info>
        </object>
      </object>
    </object>
    <object class="cMessageHeap" fullpath="simulation.scheduled-events">
      <info>length=3</info>
      <object class="cMessage" fullpath="simulation.scheduled-events.job">
        <info>src=FifoNet.fifo (id=3)  dest=FifoNet.sink (id=4)</info>
      </object>
      <object class="cMessage" fullpath="...sendMessageEvent">
        <info>at T=9.0464.., in dt=0.00817..; selfmsg for FifoNet.gen (id=2)</info>
      </object>
      <object class="cMessage" fullpath="...end-service">
        <info>at T=9.0482.., in dt=0.01; selfmsg for FifoNet.fifo (id=3)</info>
      </object>
    </object>
  </object>
</snapshot>
```

7.10.6 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.

OMNeT++ contains a mechanism that detects stack overflows. It checks the intactness of a predefined byte pattern (`0xdeadbeef`) at the stack boundary, and reports "stack violation" if it was overwritten. The mechanism usually works fine, but occasionally it can be fooled by large – and not fully used – local variables (e.g. char buffer[256]): if the byte pattern happens to fall in the middle of such a local variable, it may be preserved intact and OMNeT++ does not detect the stack violation.

To be able to make a good guess about stack size, you can use the `getStackUsage()` call which tells you how much stack the module actually uses. It is most conveniently called from `finish()`:

```
void FooModule::finish()
{
  ev << getStackUsage() << " bytes of stack used\n";
}
```

The value includes the extra stack added by the user interface library (see *extraStackforEnvir* in envir/omnetapp.h), which is currently 8K for Cmdenv and at least 16K for Tkenv. [5]

`getStackUsage()` also works by checking the existence of predefined byte patterns in the stack area, so it is also subject to the above effect with local variables.

7.11 Defining New NED Functions

It is possible to extend the NED language with new functions beyond the built-in ones. New functions are implemented in C++, and then compiled into the simulation model. When a simulation program starts up, the new functions are registered in the NED runtime, and become available for use in NED and ini files.

There are two methods to define NED functions. The `Define_NED_Function()` macro is the more flexible, preferred method of the two. `Define_NED_Math_Function()` is the older one, and it supports only certain cases. Both macros have several variants. [6]

7.11.1 Define_NED_Function()

The `Define_NED_Function()` macro lets you define new functions that can accept arguments of various data types (`bool`, `double`, `string`, etc.), supports optional arguments and also variable argument lists (variadic functions).

The macro has two variants:

```
Define_NED_Function(FUNCTION,SIGNATURE);
Define_NED_Function2(FUNCTION,SIGNATURE,CATEGORY,DESCRIPTION);
```

The two variants are basically equivalent; the only difference is that the second one allows you to specify two more parameters, `CATEGORY` and `DESCRIPTION`. These two parameters expect human-readable strings that are displayed when listing the available NED functions.

The common parameters, `FUNCTION` and `SIGNATURE` are the important ones. `FUNCTION` is the name of (or pointer to) the C++ function that implements the NED function, and `SIGNATURE` is the function signature as a string; it defines the name, argument types and return type of the NED function.

[5] The actual value is platform-dependent.

[6] Before OMNeT++ 4.2, `Define_NED_Math_Function()` was called `Define_Function()`.

You can list the available NED functions by running `opp_run` or any simulation executable with the `-h nedfunctions` option. The result will be similar to what you can see in Appendix D.

```
$ opp_run -h nedfunctions
OMNeT++ Discrete Event Simulation...
Functions that can be used in NED expressions and in omnetpp.ini:

 Category "conversion":
  double : double double(any x)
    Converts x to double, and returns the result. A boolean argument becomes
    0 or 1; a string is interpreted as number; an XML argument causes an error.
 ...
```

Seeing the above output, it should now be obvious what the `CATEGORY` and `DESCRIPTION` macro arguments are for. OMNeT++ uses the following category names: `"conversion"`, `"math"`, `"misc"`, `"ned"`, `"random/continuous"`, `"random/discrete"`, `"strings"`, `"units"`, `"xml"`. You can use these category names for your own functions as well, when appropriate.

The Signature

The signature string has the following syntax:

returntype functionname(argtype1 argname1, argtype2 argname2, ...)

The *functionname* part defines the name of the NED function, and it must meet the syntactical requirements for NED identifiers (start with a letter or underscore, not be a reserved NED keyword, etc.)

The argument types and return type can be one of the following: **`bool`**, **`int`** (maps to C/C++ `long`), **`double`**, **`quantity`**, **`string`**, **`xml`** or **`any`**; that is, any NED parameter type plus **`quantity`** and **`any`**. **`quantity`** means *double with an optional measurement unit* (**`double`** and **`int`** only accept dimensionless numbers), and **`any`** stands for any type. The argument names are presently ignored.

To make arguments optional, append a question mark to the argument name. Like in C++, optional arguments may only occur at the end of the argument list, i.e. all arguments after an optional argument must also be optional. The signature string does not have syntax for supplying default values for optional arguments; that is, default values have to be built into the C++ code that implements the NED function. To let the NED function accept any number of additional arguments of arbitrary types, add an ellipsis (`...`) to the signature as the last argument.

Some examples:

```
"int factorial(int n)"
"bool isprime(int n)"
"double sin(double x)"
"string repeat(string what, int times)"
"quantity uniform(quantity a, quantity b, long rng?)"
"any choose(int index, ...)"
```

The first three examples define NED functions with the names `factorial`, `isprime` and `sin`, with the obvious meanings. The fourth example can be the signature for a function that repeats a string n times, and returns the concatenated result. The fifth example is

the signature of the existing `uniform()` NED function; it accepts numbers both with and without measurement units (of course, when invoked with measurement units, both `a` and `b` must have one, and the two must be compatible – this should be checked by the C++ implementation). `uniform()` also accepts an optional third argument, an RNG index. The sixth example can be the signature of a `choose()` NED function that accepts an integer plus any number of additional arguments of any type, and returns the *index*th one among them.

Implementing the NED Function

The C++ function that implements the NED function must have the following signature, as defined by the `NEDFunction` typedef:

```
cNEDValue function(cComponent *context, cNEDValue argv[], int argc);
```

As you can see, the function accepts an array of `cNEDValue` objects, and returns a `cNEDValue`; the *argc-argv* style argument list should be familiar to you from the declaration of the C/C++ `main()` function. `cNEDValue` is a class that is used during the evaluation of NED expressions, and represents a value together with its type. The `context` argument contains the module or channel in the context of which the NED expression is being evaluated; it is useful for implementing NED functions like `getParentModuleIndex()`.

The function implementation does not need to worry too much about checking the number and types of the incoming arguments, because the NED expression evaluator already does that: inside the function you can be sure that the number and types of arguments correspond to the function signature string. Thus, `argc` is mostly useful only if you have optional arguments or a variable argument list. The NED expression evaluator also checks that the value you return from the function corresponds to the signature.

`cNEDValue` can store all the needed data types (`bool`, `double`, `string`, etc.), and is equipped with the functions necessary to conveniently read and manipulate the stored value. The value can be read via functions like `boolValue()`, `longValue()`, `doubleValue()`, `stringValue()` (returns `const char *`), `stdstringValue()` (returns `const std::string&`) and `xmlValue()` (returns `cXMLElement*`), or by simply casting the object to the desired data type, making use of the provided typecast operators. Invoking a getter or typecast operator that does not match the stored data type will result in a runtime error. For setting the stored value, `cNEDValue` provides a number of overloaded `set()` functions, assignment operators and constructors.

Further `cNEDValue` member functions provide access to the stored data type; yet other functions are associated with handling quantities, i.e. doubles with measurement units. There are member functions for getting and setting the number part and the measurement unit part separately; for setting the two components together; and for performing unit conversion.

Equipped with the above information, we can already write a simple NED function that returns the length of a string:

```
static cNEDValue ned_strlen(cComponent *context, cNEDValue argv[], int argc)
{
    return (long)argv[0].stdstringValue().size();
}

Define_NED_Function(ned_strlen, "int length(string s)");
```

Note that since `Define_NED_Function()` expects the C++ function to be already declared, we place the function implementation in front of the `Define_NED_Function()` line. We also

declare the function to be `static`, because its name doesn't need to be visible for the linker. In the function body, we use `std::string`'s `size()` method to obtain the length of the string, and cast the result to `long`; the C++ compiler will convert that into a `cNEDValue` using `cNEDValue`'s `long` constructor. Note that the **`int`** keyword in the signature maps to the C++ type `long`.

The following example defines a `choose()` NED function that returns its kth argument that follows the `index` (k) argument.

```
static cNEDValue ned_choose(cComponent *context, cNEDValue argv[], int argc)
{
    int index = (int)argv[0];
    if (index < 0 || index >= argc-1)
        throw cRuntimeError("choose(): index %d is out of range", index);
    return argv[index+1];
}

Define_NED_Function(ned_choose, "any choose(int index, ...)");
```

Here, the value of `argv[0]` is read using the typecast operator that maps to `longValue()`. (Note that if the value of the `index` argument does not fit into an `int`, the conversion will result in data loss!) The code also shows how to report errors (by throwing a `cRuntimeError`.)

The third example shows how the built-in `uniform()` NED function could be reimplemented by the user:

```
static cNEDValue ned_uniform(cComponent *context, cNEDValue argv[], int argc)
{
    int rng = argc==3 ? (int)argv[2] : 0;
    double argv1converted = argv[1].doubleValueInUnit(argv[0].getUnit());
    double result = uniform((double)argv[0], argv1converted, rng);
    return cNEDValue(result, argv[0].getUnit());
    // or: argv[0].setPreservingUnit(result); return argv[0];
}

Define_NED_Function(ned_uniform, "quantity uniform(quantity a, quantity b, int rng?)"
```

The first line of the function body shows how to supply default values for optional arguments; for the `rng` argument in this case. The next line deals with unit conversion. This is necessary because the `a` and `b` arguments are both quantities and may come in with different measurement units. We use the `doubleValueInUnit()` function to obtain the numeric value of `b` in `a`'s measurement unit. If the two units are incompatible or only one of the parameters have a unit, an error will be raised. If neither parameters have a unit, `doubleValueInUnit()` simply returns the stored `double`. Then we call the `uniform()` C++ function to actually generate a random number, and return it in a temporary object with `a`'s measurement unit. Alternatively, we could have overwritten the numeric part of `a` with the result using `setPreservingUnit()`, and returned just that. If there is no measurement unit, `getUnit()` will return `NULL`, which is understood by both `doubleValueInUnit()` and the `cNEDValue` constructor.

NOTE: Note that it is OK to change the elements of the `argv[]` vector: they will be discarded (popped off the evaluation stack) by the NED expression evaluator anyway when your function returns.

cNEDValue In More Detail

In the previous section we have given an overview and demonstrated the basic use of the `cNEDValue` class; here we go into further details.

The stored data type can be obtained with the `getType()` function. It returns an enum (`cNEDValue::Type`) that has the following values: `UNDEF`, `BOOL`, `DBL`, `STR`, `XML`. `UNDEF` is synonymous with *unset*; the others have the obvious meanings. There is no separate `QUANTITY` type: quantities are also represented with the `DBL` type, which has an optional associated measurement unit. Note that `LONG` is also missing; the reason is that the NED expression evaluator currently (as of OMNeT++ 4.2) stores all numbers as `double`s. [7]

The `getTypeName()` static function returns the string equivalent of a `cNEDValue::Type`. The utility functions `isSet()` and `isNumeric()` check that the type is (not) `UNDEF` and `DBL`, respectively.

```
cNEDValue value = 5.0;
cNEDValue::Type type = value.getType(); // ==> DBL
ev << cNEDValue::getTypeName(type) << endl; // ==> "double"
```

We have already seen that the `DBL` type serves both the **`double`** and **`quantity`** types of the NED function signature, by storing an optional measurement unit (a string) in addition to the `double` variable. A `cNEDValue` can be set to a quantity by creating it with a two-argument constructor that accepts a `double` and a `const char *` for unit, or by invoking a similar two-argument `set()` function. The measurement unit can be read with `getUnit()`, and overwritten with `setUnit()`. If you assign a `double` to a `cNEDValue` or invoke the one-argument `set(double)` method on it, that will clear the measurement unit. If you want to overwrite the number part but preserve the original unit, you need to use the `setPreservingUnit(double)` method.

There are several functions that perform unit conversion. The `doubleValueInUnit()` method accepts a measurement unit, and attempts to return the number in that unit. The `convertTo()` method also accepts a measurement unit, and tries to permanently convert the value to that unit; that is, if successful, it changes both the number and the measurement unit part of the object. The `convertUnit()` static `cNEDValue` member function accepts three arguments: a quantity as a `double` and a unit, and a target unit; and returns the number in the target unit. A `parseQuantity()` static member function parses a string that contains a quantity (e.g. `"5min 48s"`), and return both the numeric value and the measurement unit. Another version of `parseQuantity()` tries to return the value in a unit you specify. All functions raise an error if the unit conversion is not possible, e.g. due to incompatible units.

For performance reasons, `setUnit()`, `convertTo()` and all other functions that accept and store a measurement unit will only store the `const char*` pointer, but do *not* copy the string itself. Consequently, the passed measurement unit pointers must stay valid for at least the lifetime of the `cNEDValue` object, or even longer if the same pointer propagates to other `cNEDValue` objects. It is recommended that you only pass pointers that stay valid during the entire simulation. It is safe to use: (1) string constants from the code; (2) unit strings from other `cNEDValue`s; and (3) pooled strings e.g. from a `cStringPool` or from `cNEDValue`'s static `getPooled()` function.

Example code:

```
// manipulating the number and the measurement unit
```

[7] The IEEE *double*'s mantissa is 53 bits, so `double` can accurately represent 32-bit integers, the usual size of `long` on 32-bit architectures. On 64-bit architectures the usual size of `long` is 64 bits, so precision loss will occur when converting very large integers to `double`. Note, however, that simulations that trigger this precision loss would not be able to run on 32-bit architectures at all!

```
cNEDValue value(250,"ms");     // initialize to 250ms
value = 300.0;                 // ==> 300 (clears the unit!)
value.set(500,"ms");           // ==> 500ms
value.setUnit("s");            // ==> 500s (overwrites the unit)
value.setPreservingUnit(180); // ==> 180s (overwrites the number)
value.setUnit(NULL);           // ==> 180 (clears the unit)

// unit conversion
value.set(500, "ms");          // ==> 500ms
value.convertTo("s");          // ==> 0.5s
double us = value.doubleValueInUnit("us"); // ==> 500000 (value is unchanged)
double bps = cNEDValue::convertUnit(128, "kbps", "bps"); // ==> 128000
double ms = cNEDValue::convertUnit("2min 15.1s", "ms"); // ==> 135100

// getting persistent measurement unit strings
const char *unit = argv[0].stringValue(); // cannot be trusted to persist
value.setUnit(cNEDValue::getPooled(unit)); // use a persistent copy for setUnit()
```

7.11.2 Define_NED_Math_Function()

The `Define_NED_Math_Function()` macro lets you register a C/C++ "mathematical" function as a NED function. The registered C/C++ function may take up to four `double` arguments, and must return a `double`; the NED signature will be the same. In other words, functions registered this way cannot accept any NED data type other than `double`; cannot return anything else than `double`; cannot accept or return values with measurement units; cannot have optional arguments or variable argument lists; and are restricted to four arguments at most. In exchange for these restrictions, the C++ implementation of the functions is a lot simpler.

Accepted function signatures for `Define_NED_Math_Function()`:

```
double f();
double f(double);
double f(double, double);
double f(double, double, double);
double f(double, double, double, double);
```

The simulation kernel uses `Define_NED_Math_Function()` to expose commonly used `<math.h>` functions in the NED language. Most `<math.h>` functions (`sin()`, `cos()`, `fabs()`, `fmod()`, etc.) can be directly registered without any need for wrapper code, because their signatures is already one of the accepted ones listed above.

The macro has the following variants:

```
Define_NED_Math_Function(NAME,ARGCOUNT);
Define_NED_Math_Function2(NAME,FUNCTION,ARGCOUNT);
Define_NED_Math_Function3(NAME,ARGCOUNT,CATEGORY,DESCRIPTION);
Define_NED_Math_Function4(NAME,FUNCTION,ARGCOUNT,CATEGORY,DESCRIPTION);
```

All macros accept the `NAME` and `ARGCOUNT` parameters; they are the intended name of the NED function and the number of `double` arguments the function takes (0..3). `NAME` should be provided without quotation marks (they will be added inside the macro.) Two macros also accept a `FUNCTION` parameter, which is the name of (or pointer to) the implementation C/C++ function. The macros that don't have a `FUNCTION` parameter simply use the `NAME` parameter

for that as well. The last two macros accept `CATEGORY` and `DESCRIPTION`, which have exactly the same role as with `Define_NED_Function()`.

Examples:

```
Define_NED_Math_Function3(sin, 1, "math", "Trigonometric function; see <math.h>");
Define_NED_Math_Function3(cos, 1, "math", "Trigonometric function; see <math.h>");
Define_NED_Math_Function3(pow, 2, "math", "Power-of function; see <math.h>");
```

7.12 Deriving New Classes

7.12.1 cOwnedObject or Not?

If you plan to implement a completely new class (as opposed to subclassing something already present in OMNeT++), you have to ask yourself whether you want the new class to be based on `cOwnedObject` or not. Note that we are *not* saying you should always subclass from `cOwnedObject`. Both solutions have advantages and disadvantages, which you have to consider individually for each class.

`cOwnedObject` already carries (or provides a framework for) significant functionality that is either relevant to your particular purpose or not. Subclassing `cOwnedObject` generally means you have more code to write (as you *have to* redefine certain virtual functions and adhere to conventions) and your class will be a bit more heavy-weight. However, if you need to store your objects in OMNeT++ objects like `cQueue` or you want to store OMNeT++ classes in your object, then you *must* subclass from `cOwnedObject`. [8]

The most significant features of `cOwnedObject` are the name string (which has to be stored somewhere, so it has its overhead) and ownership management (see section 7.13), which also provides advantages at some cost.

As a general rule, small `struct`-like classes like `IPAddress`, `MACAddress`, `RoutingTableEntry`, `TCPConnectionDescriptor`, etc. are better *not* subclassed from `cOwnedObject`. If your class has at least one virtual member function, consider subclassing from `cObject`, which does not impose any extra cost because it doesn't have data members at all, only virtual functions.

7.12.2 cOwnedObject Virtual Methods

Most classes in the simulation class library are descendants of `cOwnedObject`. If you want to derive a new class from `cOwnedObject` or a `cOwnedObject` descendant, you must redefine some member functions so that objects of the new type can fully co-operate with other parts of the simulation system. A more or less complete list of these functions is presented here. You don't need to worry about the length of the list: most functions are not absolutely necessary to implement. For example, you do not need to redefine `forEachChild()` unless your class is a container class.

The following methods **must** be implemented:

- *Constructor.* At least two constructors should be provided: one that takes the object name string as `const char *` (recommended by convention), and another one with no

[8] For simplicity, in these sections "OMNeT++ object" should be understood as "object of a class subclassed from `cOwnedObject`"

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&)`.
- *Destructor*.
- *Duplication function*, `X *dup() const`. It should create and return an exact duplicate of the object. It is usually a one-line function that delegates to the copy constructor.
- *Assignment operator*, that is, `X& operator=(const X&)` for a class `X`. It should copy the contents of the other object into this one, *except* the name string. See later what to do if the object contains pointers to other objects.

If your class contains other objects subclassed from `cOwnedObject`, either via pointers or as a data member, the following function **should** be implemented:

- *Iteration function*, `void forEachChild(cVisitor *v)`. The implementation should call the function passed for each object it contains via pointer or as a data member; see the API Reference on `cOwnedObject` on how to implement `forEachChild()`. `forEachChild()` makes it possible for Tkenv to display the object tree to you, to perform searches on it, etc. It is also used by `snapshot()` and some other library functions.

Implementation of the following methods is **recommended**:

- *Object info*, `std::string info()`. The `info()` function should return a one-line string describing the object's contents or state. `info()` is displayed at several places in Tkenv.
- *Detailed object info*, `std::string detailedInfo()`. This method may potentially be implemented in addition to `info()`; it can return a multi-line description. `detailedInfo()` is also displayed by Tkenv in the object's inspector.
- *Serialization*, `parsimPack()` and `parsimUnpack()` methods. These methods are needed for parallel simulation, if you want objects of this type to be transmitted across partitions.

It is customary to implement the copy constructor and the assignment operator so that they delegate to the same function of the base class, and invoke a common private `copy()` function to copy the local members.

7.12.3 Class Registration

You should also use the `Register_Class()` macro to register the new class. It is used by the `createOne()` factory function, which can create any object given the class name as a string. `createOne()` is used by the Envir library to implement `omnetpp.ini` options such as `rng-class="..."` or `scheduler-class="..."`. (see Chapter 16)

For example, an `omnetpp.ini` entry such as

```
rng-class = "cMersenneTwister"
```

would result in something like the following code to be executed for creating the RNG objects:

```
cRNG *rng = check_and_cast<cRNG*>(createOne("cMersenneTwister"));
```

But for that to work, we needed to have the following line somewhere in the code:

```
Register_Class(cMersenneTwister);
```

`createOne()` is also needed by the parallel distributed simulation feature (Chapter 15) to create blank objects to unmarshal into on the receiving side.

7.12.4 Details

We'll go through the details using an example. We create a new class `NewClass`, redefine all above mentioned `cOwnedObject` member functions, and explain the conventions, rules and tips associated with them. To demonstrate as much as possible, the class will contain an `int` data member, dynamically allocated non-`cOwnedObject` data (an array of `double`s), an OMNeT++ object as data member (a `cQueue`), and a dynamically allocated OMNeT++ object (a `cMessage`).

The class declaration is the following. It contains the declarations of all methods discussed in the previous section.

```
//
// file: NewClass.h
//
#include <omnetpp.h>

class NewClass : public cOwnedObject
{
  protected:
    int size;
    double *array;
    cQueue queue;
    cMessage *msg;
    ...
  private:
    void copy(const NewClass& other); // local utility function
  public:
    NewClass(const char *name=NULL, int d=0);
    NewClass(const NewClass& other);
    virtual ~NewClass();
    virtual NewClass *dup() const;
    NewClass& operator=(const NewClass& other);

    virtual void forEachChild(cVisitor *v);
    virtual std::string info();
};
```

We'll discuss the implementation method by method. Here is the top of the `.cc` file:

```
//
// file: NewClass.cc
//
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include "newclass.h"
```

```
Register_Class( NewClass );


NewClass::NewClass(const char *name, int sz) : cOwnedObject(name)
{
    size = sz;
    array = new double[size];
    take(&queue);
    msg = NULL;
}
```

The constructor (above) calls the base class constructor with the name of the object, then initializes its own data members. You need to call `take()` for `cOwnedObject`-based data members.

```
NewClass::NewClass(const NewClass& other) : cOwnedObject(other)
{
    size = -1; // needed by copy()
    array = NULL;
    msg = NULL;
    take(&queue);
    copy(other);
}
```

The copy constructor relies on the private `copy()` function. Note that pointer members have to be initialized (to `NULL` or to an allocated object/memory) before calling the `copy()` function.

You need to call `take()` for `cOwnedObject`-based data members.

```
NewClass::~NewClass()
{
    delete [] array;
    if (msg->getOwner()==this)
        delete msg;
}
```

The destructor should delete all data structures the object allocated. `cOwnedObject`-based objects should *only* be deleted if they are owned by the object – details will be covered in section 7.13.

```
NewClass *NewClass::dup() const
{
    return new NewClass(*this);
}
```

The `dup()` function is usually just one line, like the one above.

```
NewClass& NewClass::operator=(const NewClass& other)
{
    if (&other==this)
        return *this;

    cOwnedObject::operator=(other);
    copy(other);
    return *this;
}
```

The assignment operator (above) first makes sure that will not try to copy the object to itself, because that can be disastrous. If so (that is, `&other==this`), the function returns immediately without doing anything.

The base class part is copied via invoking the assignment operator of the base class. Then the method copies over the local members using the `copy()` private utility function.

```
void NewClass::copy(const NewClass& other)
{
    if (size != other.size) {
        size = other.size;
        delete array;
        array = new double[size];
    }
    for (int i = 0; i < size; i++)
        array[i] = other.array[i];

    queue = other.queue;
    queue.setName(other.queue.getName());

    if (msg && msg->getOwner()==this)
        delete msg;

    if (other.msg && other.msg->getOwner()==const_cast<cMessage*>(&other))
        take(msg = other.msg->dup());
    else
        msg = other.msg;
}
```

Complexity associated with copying and duplicating the object is concentrated in the `copy()` utility function.

Data members are copied in the normal C++ way. If the class contains pointers, you will most probably want to make a deep copy of the data where they point, and not just copy the pointer values.

If the class contains pointers to 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::forEachChild(cVisitor *v)
{
    v->visit(queue);
    if (msg)
        v->visit(msg);
}
```

The `forEachChild()` function should call `v->visit(obj)` for each `obj` member of the class. See the API Reference for more information about `forEachChild()`.

```
std::string NewClass::info()
{
    std::stringstream out;
    out << "data=" << data << ", array[0]=" << array[0];
    return out.str();
```

```
}
```

The `info()` method should produce a concise, one-line string about the object. You should try not to exceed 40-80 characters, since the string will be shown in tooltips and listboxes.

See the virtual functions of `cObject` and `cOwnedObject` in the class library reference for more information. The sources of the Sim library (`include/`, `src/sim/`) can serve as further examples.

7.13 Object Ownership Management

7.13.1 The Ownership Tree

OMNeT++ has a built-in ownership management mechanism which is used for sanity checks, and as part of the infrastructure supporting Tkenv inspectors.

Container classes like `cQueue` own the objects inserted into them, but this is not limited to objects inserted into a container: *every `cOwnedObject`-based object has an owner all the time*. From the user's point of view, ownership is managed transparently. For example, when you create a new `cMessage`, it will be owned by the simple module. When you send it, it will first be handed over to (i.e. change ownership to) the FES, and, upon arrival, to the destination simple module. When you encapsulate the message in another one, the encapsulating message will become the owner. When you decapsulate it again, the currently active simple module becomes the owner.

The `getOwner()` method, defined in `cOwnedObject`, returns the owner of the object:

```
cOwnedObject *o = msg->getOwner();
ev << "Owner of " << msg->getName() << " is: " <<
   << "(" << o->getClassName() << ") " << o->getFullPath() << endl;
```

The other direction, enumerating the objects owned can be implemented with the `forEachChild()` method by it looping through all contained objects and checking the owner of each object.

Why Do We Need This?

The traditional concept of object ownership is associated with the "right to delete" objects. In addition to that, keeping track of the owner and the list of objects owned also serves other purposes in OMNeT++:

- enables methods like `getFullPath()` to be implemented.
- prevents certain types of programming errors, namely, those associated with wrong ownership handling.
- enables Tkenv to display the list of simulation objects present within a simple module. This is extremely useful for finding memory leaks caused by forgetting to delete messages that are no longer needed.

Some examples of programming errors that can be caught by the ownership facility:

- attempts to send a message while it is still in a queue, encapsulated in another message, etc.
- attempts to send/schedule a message while it is still owned by the simulation kernel (i.e. scheduled as a future event)
- attempts to send the very same message object to multiple destinations at the same time (ie. to all connected modules)

For example, the `send()` and `scheduleAt()` functions check that the message being sent/scheduled is owned by the module. If it is not, then it signals a programming error: the message is probably owned by another module (already sent earlier?), or currently scheduled, or inside a queue, a message or some other object – in either case, the module does not have any authority over it. When you get the error message (`"not owner of object"`), you need to carefully examine the error message to determine which object has ownership of the message, and correct the logic that caused the error.

The above errors are easy to make in the code, and if not detected automatically, they could cause random crashes which are usually very difficult to track down. Of course, some errors of the same kind still cannot be detected automatically, like calling member functions of a message object which has been sent to (and so is currently owned by) another module.

7.13.2 Managing Ownership

Ownership is managed transparently for the user, but this mechanism has to be supported by the participating classes themselves. It will be useful to look inside `cQueue` and `cArray`, because they might give you a hint what behavior you need to implement when you want to use non-OMNeT++ container classes to store messages or other `cOwnedObject`-based objects.

Insertion

`cArray` and `cQueue` have internal data structures (array and linked list) to store the objects which are inserted into them. However, they do *not* necessarily own all of these objects. (Whether they own an object or not can be determined from that object's `getOwner()` pointer.)

The default behaviour of `cQueue` and `cArray` is to take ownership of the objects inserted. This behavior can be changed via the *takeOwnership* flag.

Here is what the *insert* operation of `cQueue` (or `cArray`) does:

- insert the object into the internal array/list data structure
- if the *takeOwnership* flag is true, take ownership of the object, otherwise just leave it with its original owner

The corresponding source code:

```
void cQueue::insert(cOwnedObject *obj)
{
    // insert into queue data structure
    ...

    // take ownership if needed
    if (getTakeOwnership())
```

```
        take(obj);

}
```

Removal

Here is what the *remove* family of operations in `cQueue` (or `cArray`) does:

- remove the object from the internal array/list data structure
- if the object is actually owned by this `cQueue`/`cArray`, release ownership of the object, otherwise just leave it with its current owner

After the object was removed from a `cQueue`/`cArray`, you may further use it, or if it is not needed any more, you can delete it.

The *release ownership* phrase requires further explanation. When you remove an object from a queue or array, the ownership is expected to be transferred to the simple module's local objects list. This is accomplished by the `drop()` function, which transfers the ownership to the object's default owner. `getDefaultOwner()` is a virtual method returning `cOwnedObject*` defined in `cOwnedObject`, and its implementation returns the currently executing simple module's local object list.

As an example, the `remove()` method of `cQueue` is implemented like this: [9]

```
cOwnedObject *cQueue::remove(cOwnedObject *obj)
{
    // remove object from queue data structure
    ...

    // release ownership if needed
    if (obj->getOwner()==this)
        drop(obj);

    return obj;
}
```

Destructor

The concept of `ownership` is that *the owner has the exclusive right and duty to delete the objects it owns*. For example, if you delete a `cQueue` containing `cMessage`s, all messages it contains *and* owns will also be deleted.

The destructor should delete all data structures the object allocated. From the contained objects, only the owned ones are deleted – that is, where `obj->getOwner()==this`.

Object Copying

The ownership mechanism also has to be taken into consideration when a `cArray` or `cQueue` object is duplicated (using `dup()` or the copy constructor.) The duplicate is supposed to

[9] Actual code in `src/sim` is structured somewhat differently, but the meaning is the same.

have the same content as the original; however, the question is whether the contained objects should also be duplicated or only their pointers taken over to the duplicate `cArray` or `cQueue`. A similar question arises when an object is copied using the assignment operator (`operator=()`).

The convention followed by `cArray`/`cQueue` is that only owned objects are copied, and the contained but not owned ones will have their pointers taken over and their original owners left unchanged.

Chapter 8

Network Graphics And Animation

8.1 Display Strings

Display strings specify the arrangement and appearance of representations of modules in graphical user interfaces (currently only Tkenv); they control how the objects (compound modules, their submodules and connections) are displayed. Display strings are specified in NED's `@display` property.

Display strings can be used in the following contexts:

- *submodules* – display strings may contain position, arrangement (for module vectors), icon, icon color, auxiliary icon, status text, communication range (as circle or filled circle), tooltip, etc.
- *compound modules, networks* – display strings can specify background color, border color, border width, background image, scaling, grid, unit of measurement, etc.
- *connections* – display strings can specify positioning, color, line width, line style, text and tooltip
- *messages* – display strings can specify icon, icon color, etc.

8.1.1 Display String Syntax

The display string syntax is a semicolon-separated list of tags. Each tag consists of a key, an equal sign and a comma-separated list of arguments:

```
@display("p=100,100;b=60,10,rect,blue,black,2")
```

Tag arguments may be omitted both at the end and inside the parameter list. If an argument is omitted, a sensible default value is used.

```
@display("p=100,100;b=,,rect,blue")
```

8.1.2 Display String Placement

The following NED sample shows where to place display strings in the code:

```
simple Queue
{
    parameters:
        @display("i=block/queue");
    ...
}

network SimpleQueue
{
    parameters:
        @display("bgi=maps/europe");
    submodules:
        sink: Sink {
            @display("p=273,101");
        }
        ...
    connections:
        source.out --> { @display("ls=red,3"); } --> queue.in++;
}
```

8.1.3 Display String Inheritance

Every module and channel object has one single display string object, which controls its appearance in various contexts. The initial value of this display string object comes from merging the `@display` properties occurring at various places in NED files. This section describes the rules for merging `@display` properties to create the module or channel's display string.

- Derived NED types inherit their display string from their base NED type.
- Submodules inherit their display string from their type.
- Connections inherit their display string from their channel type.

The base NED type's display string is merged into the current display string using the following rules:

- If a tag is present in the base display string, but not in the current one the whole tag (with all arguments) is added to the current display string. (e.g. base: `"i=icon,red"` current: `"p=2,4"` result: `"p=2,4;i=icon,red"`)
- If a tag is present both in the base and in the current display string only tag arguments present in the base, but not in the current display string will be copied. (e.g. base: `"b=40,20"` current: `"b=,30,oval"` result: `"b=40,30,oval"`)
- If the current display string contains a tag argument with value "-" (hyphen) that argument is treated as empty and will not be inherited from other display strings. Requesting the value of this argument will return its the default value.
- If neither the base display string nor the current one has value for a tag a suitable default value will be returned and used.

Example of display string inheritance:

```
simple Base {
    @display("i=block/queue"); // use a queue icon in all instances
}

simple Derived extends Base {
    @display("i=,red,60");  // ==> "i=block/queue,red,60"
}

network SimpleQueue {
    submodules:
        submod: Derived {
            @display("i=,yellow,-;p=273,101;r=70");
                    // ==> "i=block/queue,yellow;p=273,101;r=70"
        }
        ...
}
```

8.1.4 Display String Tags Used in Submodule Context

The following tags define how a module appears on the Tkenv user interface if it is used as a submodule:

- `b` – shapes, colors
- `i` – icon
- `is` – icon size
- `i2` – alternate (status) icon placed at the upper right corner of the main icon
- `p` – positioning and layout
- `r` – range indicator
- `q` – queue information text
- `t` – text
- `tt` – tooltip

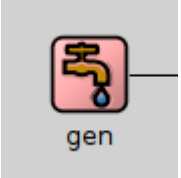
Icons

By default, modules are represented by simple icons. Using images for the modules is possible with the `i` tag. See the `images` subfolder of your OMNeT++ installation for possible icons. The stock images installed with OMNeT++ have several size variants. Most of them have very small (vs), small (s), large (l) and very large (vl) variants. You can specify which variant you want to use with the `is` tag.

```
@display("i=block/source;is=l"); // a large source icon from the block icons group
```

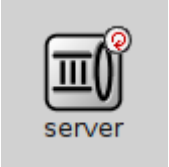
Sometimes you want to have similar icons for modules, but would like to make them look a little different to create groups or to reflect status information about the module. You can easily change the color of an already existing image. The following example colorizes the `block/source` icon, 20% red

```
@display("i=block/source,red,20")
```



If you want to show state information about your module, you can use the `i2` tag to add a small status icon to your main icon. This icon is displayed in the upper right corner of your main icon. In most cases the `i2` tag is specified at runtime using the `setDisplayString()` method, so the icon can be changed dynamically based on the module's internal state.

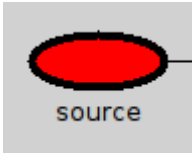
```
@display("i=block/queue;i2=status/busy")
```



Shapes

If you want to have simple, but resizable representation for your module, you can use the `b` tag to create geometric shapes. Currently `oval` and `rectangle` is supported:

```
// an oval shape with 70x30 size, red background, black 4 pixel border
@display("b=70,30,oval,red,black,4")
```



Positioning, Coordinates

To define the position of a module inside an other one use the `p` tag. If you do not specify a `p` tag for your module, the parent module will automatically choose a position based on a layout algorithm. The following example will place the module at the given position:

```
@display("p=50,79");
```

NOTE: The coordinates specified in the `p`, `b` or `r` tags are not necessarily integers and measured in pixels. You can use the parent module's `bgs=`*pix2unitratio*`,`*unit* tag, to set the scaling parameter and the unit of measurement for your module. You can specify the ratio between 1 pixel and 1 unit with the `bgs` tag.

The `p` tag allows the automatic arrangement of module vectors. They can be arranged in a row, a column, a matrix or a ring, or you may specify their positions later at runtime using the `setDisplayString()` method. The rest of the arguments in the `p` tag depend on the layout type:

- `row` – `p=100,100,r,`$deltaX$ (A row of modules with $deltaX$ units between the modules)
- `column` – `p=100,100,c,`$deltaY$ (A column of modules with $deltaX$ units between the modules)
- `matrix` – `p=100,100,m,`$noOfCols$`,`$deltaX$`,`$deltaY$ (A matrix with $noOfCols$ columns. $deltaX$ and $deltaY$ units between rows and columns)
- `ring` – `p=100,100,ri,`rx`,`ry (A ring (oval) with rx and ry as the horizontal and vertical radius.)
- `exact (default)` – `p=100,100,x,`$deltaX$`,`$deltaY$ (Place each module at $(100+deltaX, 100+deltaY)$. The coordinates are usually set at runtime.)

A matrix layout for a module vector:

```
@display("p=,,m,4,50,50");
```

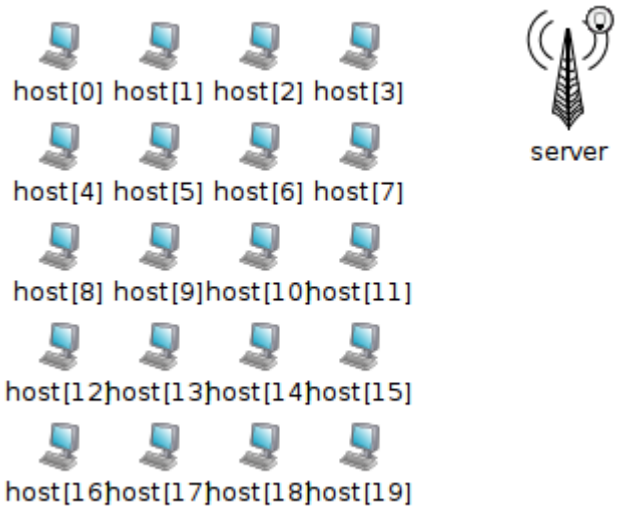


Figure 8.1: Matrix arrangement using the p tag

Wireless Range

In wireless simulations it is very useful to show some kind of range around your module. This can be an interference range, transmission range etc. The following example will place the module at a given position, and draw a circle with a 90-unit radius around it as a range indicator:

```
submodules:
    ap: AccessPoint {
        @display("p=50,79;r=90");
    }
```

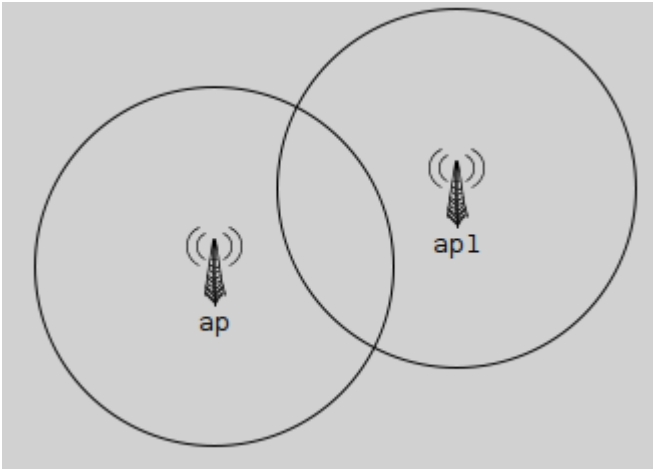
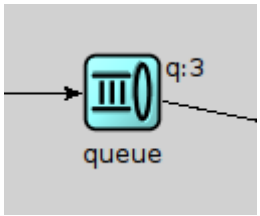


Figure 8.2: Range indicator using the r tag

Additional Decorations

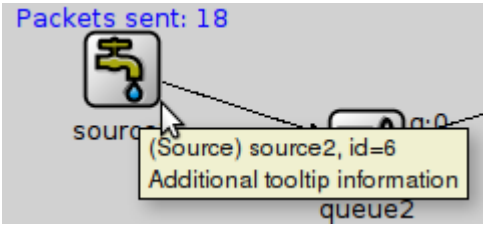
Sometimes you want to annotate your modules with additional information to make your model more transparent. One special case is when you want to show the length of a queue (`cQueue`) embedded somewhere in a module. In the following example the Server simple module contains a `cQueue` object, which was named by the `queue.setName("procqueue")` method. If we specify `q=procqueue` in the display string, Tkenv will descend into the module (several levels deep if needed) and look for a queue object named "procqueue". It will display the length of the queue object along the module.

```
@display("q=procqueue");
```



You can add a text description to any module using the `t` (displayed along the module) or `tt` tag (displayed as a tooltip). The following example displays a short text string along with the module and adds a tooltip text string that can be seen by hovering over them module with the mouse.

```
@display("t=Packets sent: 18;tt=Additional tooltip information");
```



NOTE: The `t` and `tt` tags, when set at runtime, can be used to display various information about the module's state. The `setTagArg()` method of `cDisplayString` can be used to update the text:

```
char buf[64];
sprintf(buf, "sent: %d, rcvd: %d", numPkSent, numPkReceived);
getDisplayString().setTagArg("t", 0, buf);
```

For a detailed descripton of the display string tags, check Appendix F.

8.1.5 Display String Tags Used in Module Background Context

The following tags describe what a module looks like when opened in Tkenv. They mostly deal with the module background.

- `bgi` – background image
- `bgtt` – tooltip above the background
- `bgg` – background grid
- `bgl` – child layout
- `bgb` – background size, color, border
- `bgs` – scaling of background coordinates
- `bgp` – background coordinate offset

The `bgs` tag makes it possible to use a physical unit of measurement, (e.g. kilometers) for coordinates. `bgs` arguments include a pixel-per-unit factor (for mapping coordinates to the screen), and a unit name. When `bgs` is present, all coordinates (including submodule coordinates) are interpreted in the given unit of measurement. When combined with the `bgi` (background image) and `bgg` (grid) tags, it is possible to display maps.

The following example demonstrates the use the of module background tags. The coordinates are given in km (SI unit). The `bgs=`*pixelsperunit*`,`*unit* specifies pixel/unit ratio, i.e. 1km is 0.075 pixel on the screen. The whole area is 6000x4500km (`bgb=`) and the map of Europe is used as a background and stretched to fill the module background. A light grey grid is drawn with a 1000km distance between major ticks, and 2 minor ticks per major tick (`bgg=tickdistance,minorpermajorticks,color`). See Figure 8.3.

```
network EuropePlayground
{
    @display("bgb=6000,4500;bgi=maps/europe,s;bgg=1000,2,grey95;bgs=0.075,km");
```

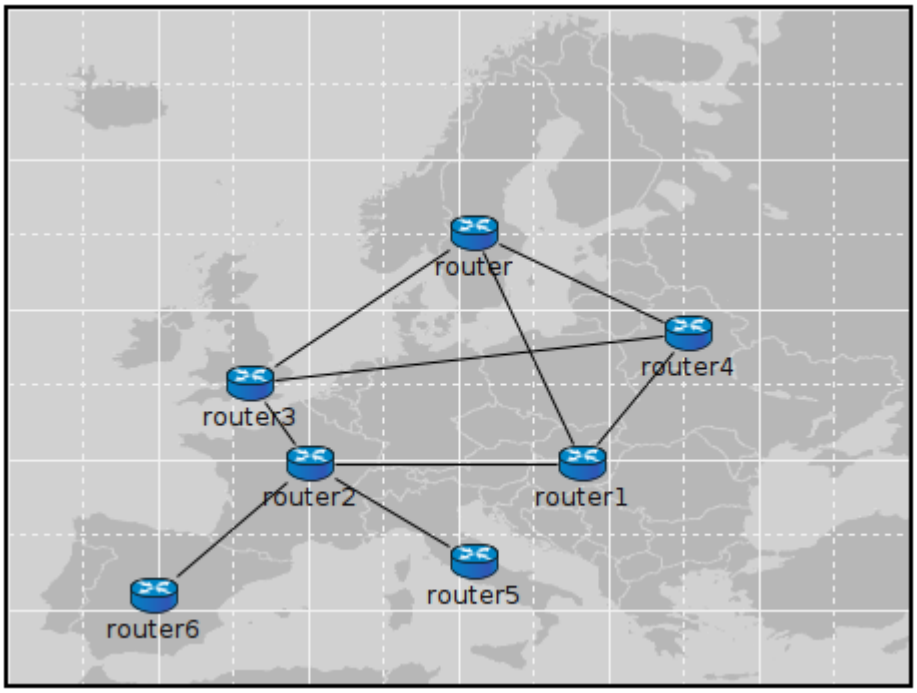


Figure 8.3: Background grid, scaling and image

After specifying the above `bgs` tag, all your submodule coordinates will be treated as if they were specified in km.

For a detailed descripton of the display string tags, check Appendix F.

8.1.6 Connection Display Strings

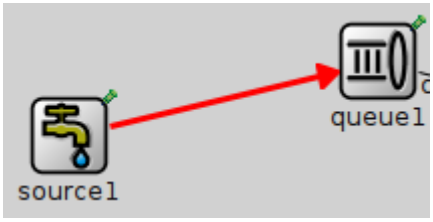
Connections may also have display strings. Connections inherit the display string property from their channel types, in the same way as submodules inherit theirs from module types. The default display strings are empty.

Connections support the following tags:

- `ls` – line style and color
- `t` – text
- `tt` – tooltip
- `m` – orientation and positioning

Example of a thick, red connection:

```
source1.out --> { @display("ls=red,3"); } --> queue1.in++;
```



NOTE: If you want to hide a connection, specify zero line width in the display string (`"ls=,0"`).

For a detailed descripton of the display string tags, check Appendix F.

8.1.7 Message Display Strings

Message display strings affect how messages are shown during animation. By default, they are displayed as a small filled circle, in one of 8 basic colors (the color is determined as *message kind modulo 8*), and with the message class and/or name displayed under it. The latter is configurable in the Tkenv Options dialog, and message kind dependent coloring can also be turned off there.

Specifying Message Display Strings

Message objects do not store a display string by default, but you can redefine the `cMessage`'s `getDisplayString()` method and make it return one.

Example of using an icon to represent a message:

```
const char *CustomPacket::getDisplayString() const
{
    return "i=msg/packet;is=vs";
}
```

Alternatively, you can add the field `displayString` to your message definition (.msg file), and the message compiler will automatically generate `setDisplayString()` and `getDisplayString()` methods for you:

```
message Job
{
    string displayString = "i=msg/package_s,kind";
...
```

Message Display String Tags

The following tags can be used in message display strings:

- `b` – shapes, colors
- `i` – icon
- `is` – icon size

Using a small red box icon to represent the messages:

```
@display("i=msd/box,red;is=s");
```

Messages will be represented by a 15x15 rectangle with white background. Their border color will depend on the `messageKind` property of the message.

```
@display("b=15,15,rect,white,kind,5");
```

NOTE: In message display strings you may use the word `kind` as a special color. This virtual color depends on the `messageKind` field in the message.

8.2 Parameter Substitution

Parameters of the module or channel containing the display string can be substituted into the display string with the `$parameterName` notation:

Example:

```
simple MobileNode
{
    parameters:
        double xpos;
        double ypos;
        string fillColor;
        // get the values from the module parameters xpos,ypos,fillcolor
        @display("p=$xpos,$ypos;b=60,10,rect,$fillColor,black,2");
}
```

8.3 Colors

8.3.1 Color Names

Any valid Tk color specification is accepted: English color names (blue, lightgrey, wheat) or *#rgb*, *#rrggbb* format (where *r*,*g*,*b* are hex digits).

It is also possible to specify colors in HSB (hue-saturation-brightness) as *@hhssbb* (with *h*, *s*, *b* being hex digits). HSB makes it easier to scale colors e.g. from white to bright red.

You can produce a transparent background by specifying a hyphen ("-") as background color.

In message display strings, `kind` can also be used as a special color name. It will map to a color depending on the message kind. (See the `getKind()` method of `cMessage`.)

8.3.2 Icon Colorization

The `"i="` display string tag allows for colorization of icons. It accepts a target color and a percentage as the degree of colorization. Percentage has no effect if the target color is missing. Brightness of the icon is also affected – to keep the original brightness, specify a color with about 50% brightness (e.g. #808080 mid-grey, #008000 mid-green).

Examples:

- "i=device/server,gold" creates a gold server icon
- "i=misc/globe,#808080,100" makes the icon greyscale
- "i=block/queue,white,100" yields a "burnt-in" black-and-white icon

Colorization works with both submodule and message icons.

8.4 Icons

8.4.1 The Image Path

In the current OMNeT++ version, module icons are PNG or GIF files. The icons shipped with OMNeT++ are in the images/ subdirectory. Both the graphical NED editor and Tkenv need the exact location of this directory to load the icons.

Icons are loaded from all directories in the *image path*, a semicolon-separated list of directories. The default image path is compiled into Tkenv with the value "*omnetpp-dir*/images;./images;./bitmaps" – which will work fine as long as you don't move the directory, and you will also be able to load more icons from the images/ subdirectory of the current directory. As users typically run simulation models from the model's directory, this practically means that custom icons placed in the images/ subdirectory of the model's directory are automatically loaded.

The compiled-in image path can be overridden with the OMNETPP_IMAGE_PATH environment variable. The way of setting environment variables is system specific: in Unix, if you are using the bash shell, adding a line

```
export OMNETPP_IMAGE_PATH="/home/you/images;./images"
```

to ~/.bashrc or ~/.bash_profile will do; on Windows, environment variables can be set via the *My Computer –> Properties* dialog.

You can extend the image path from omnetpp.ini with the tkenv-image-path option, which is prepended to the environment variable's value.

```
[General]
tkenv-image-path = "/home/you/model-framework/images;/home/you/extra-images"
```

8.4.2 Categorized Icons

Since OMNeT++ 3.0, icons are organized into several categories, represented by folders. These categories include:

- block/ - icons for subcomponents (queues, protocols, etc).
- device/ - network devices: servers, hosts, routers, etc.
- abstract/ - symbolic icons for various devices
- misc/ - node, subnet, cloud, building, town, city, etc.
- msg/ - icons that can be used for messages

Old (pre-3.0) icons are in the `old/` folder.

Tkenv and the IDE now load icons from subdirectories of all directories of the image path, and these icons can be referenced from display strings by naming the subdirectory (subdirectories) as well: `"subdir/icon"`, `"subdir/subdir2/icon"`, etc.

For compatibility, if the display string contains an icon without a category (i.e. subdirectory) name, OMNeT++ attempts to load it from "old/icon" as well.

8.4.3 Icon Size

Icons come in various sizes: normal, large, small, very small. Sizes are encoded into the icon name's suffix: `_vl`, `_l`, `_s`, `_vs`. In display strings, one can either use the suffix (`"i=device/router_l"`), or the `"is"` (*icon size*) display string tag (`"i=device/router;is=l"`), but not both at the same time (we recommend using the `is` tag whenever possible).

8.5 Layouting

OMNeT++ implements an automatic layouting feature, using a variation of the SpringEmbedder algorithm. Modules which have not been assigned explicit positions via the `"p="` tag will be automatically placed by the algorithm.

SpringEmbedder is a graph layouting algorithm based on a physical model. Graph nodes (modules) repel each other like electric charges of the same sign, and connections are sort of springs which try to contract and pull the nodes they're attached to together. There is also friction built in, in order to prevent oscillation of the nodes. The layouting algorithm simulates this physical system until it reaches equilibrium (or times out). The physical rules above have been slightly tweaked to achieve better results.

The algorithm doesn't move any module which has fixed coordinates. Predefined row, matrix, ring or other arrangements (defined via the 3rd and further args of the `"p="` tag) will be preserved – you can think about them as if those modules were attached to a rigid framework so that they can only move as one unit.

Caveats:

- If the full graph is too big after layouting, it is scaled back so that it fits on the screen, *unless it contains any fixed-position module*. (For obvious reasons: if there is a module with manually specified position, we don't want to move that one). To prevent rescaling, you can specify a sufficiently large bounding box in the background display string, e.g. `"b=2000,3000"`.

- Size is ignored by the present layouter, so longish modules (such as an Ethernet segment) may produce funny results.

- The algorithm is prone to produce erratic results, especially when the number of submodules is small, or when using predefined (matrix, row, ring, etc) layouts. The "Relayout" toobar button can then be very useful. Larger networks usually produce satisfactory results.

- The algorithm is starting from random positions. To get the best results you may experiment with different seeds by specifying them using the `bgl=`*`seed`* display string tag.

8.6 Enhancing Animation

8.6.1 Changing Display Strings at Runtime

Often it is useful to manipulate the display string at runtime. Changing colors, icon, or text may convey status change, and changing a module's position is useful when simulating mobile networks.

Display strings are stored in `cDisplayString` objects inside channels, modules and gates. `cDisplayString` also lets you manipulate the string.

To get a pointer to the `cDisplayString` object, you can call the components's `getDisplayString()` method:

```
// Setting a module's position, icon and status icon:
cDisplayString& dispStr = getDisplayString();
dispStr.parse("p=40,20;i=device/cellphone;i2=status/disconnect");
```

NOTE: The connection display string is stored in the channel object, but it can also be accessed via the source gate of the connection.

```
// Setting an outgoing connection's color to red:
cDisplayString& connDispStr = gate("out")->getDisplayString();
connDispStr.parse("ls=red");
```

NOTE: In OMNeT++ 3.x, to manipulate the appearance of a compound module you had to use the `backgroundDisplayString()` method. This method is no longer supported in OMNeT++ 4.0, because there is no separate background display string. Use the `getDisplayString()` method instead with the background specific tags, i.e. those starting with `bg`.

```
// Setting module background and grid with background display string tags:
cDisplayString& parentDispStr = getParentModule()->getDisplayString();
parentDispStr.parse("bgi=maps/europe;bgg=100,2");
```

As far as `cDisplayString` is concerned, a display string (e.g. `"p=100,125;i=cloud"`) is a string that consist of several *tags* separated by semicolons, and each tag has a *name* and after an equal sign, zero or more *arguments* separated by commas.

The class facilitates tasks such as finding out what tags a display string has, adding new tags, adding arguments to existing tags, removing tags or replacing arguments. The internal storage method allows very fast operation; it will generally be faster than direct string manipulation. The class doesn't try to interpret the display string in any way, nor does it know the meaning of the different tags; it merely parses the string as data elements separated by semicolons, equal signs and commas.

An example:

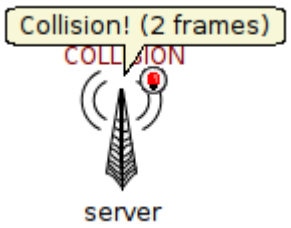
```
dispStr.parse("a=1,2;p=alpha,,3");
dispStr.insertTag("x");
dispStr.setTagArg("x",0,"joe");
dispStr.setTagArg("x",2,"jim");
dispStr.setTagArg("p",0,"beta");
ev << dispStr.str();  // result: "x=joe,,jim;a=1,2;p=beta,,3"
```

8.6.2 Bubbles

Modules can let the user know about important events (such as a node going down or coming up) by displaying a bubble with a short message ("Going down", "Coming up", etc.) This is done by the `bubble()` method of `cComponent`. The method takes the string to be displayed as a `const char *` pointer.

An example:

```
bubble("Collision! (2 frames)");
```



If the module contains a lot of code that modifies the display string or displays bubbles, it is recommended to make these calls conditional on `ev.isGUI()`. The `ev.isGUI()` call returns *false* when the simulation is run under Cmdenv, so one can make the code skip potentially expensive display string manipulation.

Better:

```
if (ev.isGUI())
    bubble("Going down!");
```

Chapter 9

Building Simulation Programs

9.1 Overview

As has already been mentioned, an OMNeT++ model consists of the following parts:

- NED language topology description(s). These are files with the `.ned` extension.
- Message definitions, in files with `.msg` extension.
- Simple module implementations and other C++ code, in `.cc` files (or `.cpp`, on Windows)

To build an executable simulation program, you first need to translate the MSG files into C++, using the message compiler (`opp_msgc`). After this step, the process is the same as building any C/C++ program from source: all C++ sources need to be compiled into object files (`.o` files (using gcc on Mac, Linux) or mingw on Windows) and all object files need to be linked with the necessary libraries to get an executable or shared library.

> **NOTE:** Compiling NED files directly to C++ code is no longer supported in OMNeT++ 4.0. NED files are always dynamically loaded.

File names for libraries differ depending on the platform, whether it is a static or shared library, and whether it is a debug or release version. Let us suppose you have a library called `common`:

- The `opp` prefix is always prepended to all library names to avoid name clashes with other programs.
- For all platforms, where the gcc (or mingw) compiler is used the static library filename has a ".a" extension and the `lib` prefix is prepended to the name (e.g. `liboppfoo[d].a`).
- For all platforms, the debug version filename has a "d" inserted at the end of the library name (e.g. `liboppcommond.ext`).
- For the Unix/Linux platform, the shared library filename has a ".so" extension (e.g. `liboppcommon[d].so`).
- For the Mac OS X platform, the shared library filename has a ".dylib" extension (e.g. `liboppcommon[d].dylib`).

- For the Windows platform, the shared library filename has a ".dll" extension (e.g. `liboppfoo[d].dll`).

In OMNeT++ 4.0 we recommend that you use shared libraries whenever it is possible. You will need to link with the following libraries:

- The simulation kernel and class library, called *oppsim* (file `liboppsim.[so|dll|dylib]` etc).
- User interfaces. The common part of all user interfaces is the *oppenvir* library (file `liboppenvir.[so|dll|dylib]`, etc), and the specific user interfaces are *opptkenv* and *oppcmdenv* (`libopptkenv.[so|dll|dylib]`, `liboppcmdenv.[so|dll|dylib]`, etc). You have to link with *oppenvir*, plus *opptkenv* or *oppcmdenv* or both.

Luckily, you do not have to worry about the above details, because automatic tools like `opp_makemake` will take care of the hard part for you.

The following figure gives an overview of the process of building and running simulation programs.

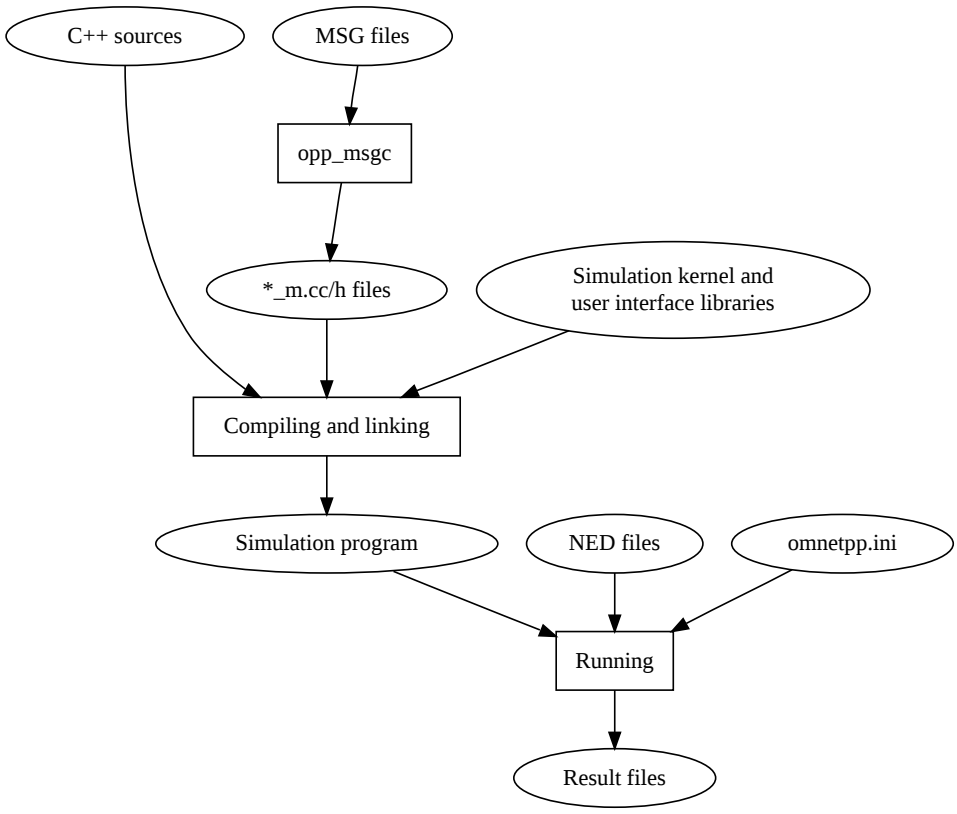


Figure 9.1: Building and running simulation

This chapter discusses how to use the simulation system on the following platforms:

- Unix (Linux/Mac OS X) with gcc
- Windows with the included MinGW compiler

9.2 Using gcc

The following section applies to using OMNeT++ on Linux, Solaris, Mac OS X, FreeBSD and other Unix derivatives, and also to MinGW on Windows.

> **NOTE:** The `doc/` directory of your OMNeT++ installation contains `Readme.<platform>` files that provide more detailed platform specific instructions.

9.2.1 The opp_makemake Tool

The `opp_makemake` tool can automatically generate a `Makefile` for your simulation program, based on the source files in the current directory or directory tree. `opp_makemake` has several options; `opp_makemake -h` displays help.

The most important options are:

- `-f, --force` : Force overwriting existing Makefile
- `-o filename` : Name of simulation executable or library to be built.
- `-O directory, --out directory` : Specifies the name of the output directory tree for out-of-directory build
- `--deep` : Generates a "deep" Makefile. A deep Makefile will cover the whole source tree under the make directory, not just files in that directory.
- `-r, --recurse` : Causes make to recursively descend into all subdirectories; subdirectories are expected to contain makefiles themselves.
- `-X directory, -Xdirectory, --except directory` : With `-r` and `--deep` option: ignore the given directory.
- `-dsubdir, -d subdir, --subdir subdir` : Causes make to recursively descend into the given directory.
- `-n, --nolink` : Produce object files but do not create executable or library.
- `-s, --make-so` : Build shared library (.so, .dll or .dylib).
- `-a, --make-lib` : Create static library (.a or .lib).
- `-Idir` : Additional NED and C++ include directory.
- `-Ldir` : Add a directory to the library path.
- `-llibrary` : Additional library to link against.

9.2.2 Basic Use

Once you have the source files (`*.ned`, `*.msg`, `*.cc`, `*.h`) in a directory, change the working directory to there and type:

```
$ opp_makemake
```

This will create a file named `Makefile`. If you type `make`, your simulation program should build.

If you already had a `Makefile` in that directory, `opp_makemake` will refuse to overwrite it. You can force overwriting the old `Makefile` with the -f option:

```
$ opp_makemake -f
```

The name of the output file will be derived from the name of the project directory (see later). You can override it with the `-o` option:

```
$ opp_makemake -f -o aloha
```

In addition to the default target that builds the simulation executable, the `Makefile` also contains the following targets:

Target	Action
all	The default target is to build the simulation executable
depend	Adds (or refreshes) dependencies in the `Makefile`
clean	Deletes all files that were produced by the make process

9.2.3 Debug and Release Builds

`opp_makemake` generates a makefile that can create both release and debug builds. By default it creates debug version, but it is easy to override this behavior. Just define the `MODE` variable on the `make` command line.

```
$ make MODE=release
```

If you want to create release builds by default you should use the `--mode` *mode* option for `opp_makemake` when generating your makefiles.

```
$ opp_makemake --mode release ...
```

9.2.4 Debugging the Makefile

`opp_makemake` generates a makefile that prints only minimal information during the build process (only the name of the compiled file.) If you want to see the full compiler commands executed by the makefile, specify `V=1` as a command line parameter for the `make` command.

```
$ make V=1
```

9.2.5 Using External C/C++ Libraries

If you are using external libraries you should specify the include path for the header files with the `-I` *includedir* option. You should specify this option if you are using anything outside of the source directory tree (except the system and OMNeT++ headers which are always included automatically)

To define an external library to be linked with, use `-Ldir` to specify the directory of the external library and `-llibrary` to specify the name of the external dependency.

9.2.6 Building Directory Trees

It is possible to build a whole source directory tree with a single makefile. A source tree will generate a single output file (executable or library). A source directory tree will always have a `Makefile` in its root, and source files may be placed anywhere in the tree.

To turn on this option, use the `opp_makemake --deep` option. `opp_makemake` will collect all `.cc` and `.msg` files from the whole subdirectory tree, and generate a makefile that covers all. If you need to exclude a specific directory, use the `-X exclude/dir/path` option. (Multiple `-X` options are accepted.)

An example:

```
$ opp_makemake -f --deep -X experimental -X obsolete
```

9.2.7 Automatic Include Dirs

If your source tree contains several subdirectories (maybe several levels deep), it can be annoying to have to specify relative paths for your header files in your `.cc` files or you should specify the include path explicitly by the `-I includepath` option. `opp_makemake` has a command line option, which adds all directories in the current source tree to the compiler command line. This option is turned on by default.

NOTE: You may turn off this mechanism with the `--no-deep-includes` option.

The only requirement is that your `#include` statements must unambigously specify the name of the header file. (i.e. if you have two `common.h` files, one in subdir1 and the other in subdir2 specify `#include "subdir1/common.h"` instead of `#include "common.h"`. If you want to include a directory which is outside of your source directory tree you always must specify it with the `-I external/include/dir` option.

9.2.8 Dependency Handling

Dependency information is used by the makefile to minimize the time required to compile and link your project. If your makefile contains up-to date dependency info – only files changed since you last compiled your project will be re-compiled or linked.

`opp_makemake` automatically adds dependencies to the makefile. You can regenerate the dependencies by typing `make depend` any time. The warnings during the dependency generation process can be safely ignored.

You may generate and add dependencies to the `makefile` manually using the `opp_makedep` tool. Use `opp_makedep --help` to display the supported command line options.

NOTE: The dependency generator does not handle conditional macros and includes. Conditionally included header files are always added to the file's dependency list.

9.2.9 Out-of-Directory Build

The build system creates object and executable files in a separate directory, called the *output directory*. The structure of the output directory will be the same as your source directory

structure except that it will be placed in the `out/configname` directory. The `configname` part will mirror your compiler toolchain and build mode settings. (i.e. The result of a debug build with gcc will be placed in `out/gcc-debug`)

The location of the generated output file is determined by the -O option. (The default value is 'out', relative to the project root directory):

```
$ opp_makemake -O ../tmp/obj
```

NOTE: The project directory is the first ancestor of the current directory which contains a `.project` file).

NOTE: Source files (i.e. those created by the `opp_msgc` compiler) will be generated in the source folder rather than in the output folder.

9.2.10 Building Shared and Static Libraries

By default the makefile will create an executable file, but it is also possible to build shared or static libraries. Shared libraries are usually a better choice.

Use `--make-so` to create shared libraries, and `--make-lib` to build static libraries. The `--nolink` option completely avoids the linking step, which is useful for top-level makefiles that only invoke other makefiles, or if you want to do the linking manually.

9.2.11 Recursive Builds

The `--recurse` option enables recursive make; when you build the simulation, make descends into the subdirectories and runs make in them too. By default, `--recurse` decends into all subdirectories; the -X directory option can be used to make it ignore certain subdirectories. This option is especially useful for top level makefiles.

The `--recurse` option automatically discovers subdirectories, but this is sometimes inconvenient. Your source directory tree may contain parts which need their own hand written `Makefile`. This can happen if you include source files from an other non OMNeT++ project. With the `-d dir` or `--subdir dir` option, you can explicitly specify which directories to recurse into, and also, the directories need not be direct children of the current directory.

The recursive make options (`--recurse`, `-d`, `--subdir`) imply `-X`, that is, the directories recursed into will be automatically excluded from deep makefiles.

You can control the order of traversal by adding dependencies into the `makefrag` file (see 9.2.12)

NOTE: With `-d`, it is also possible to create infinite recursions. `opp_makemake` cannot detect them, it is your responsibility that cycles do not occur.

Motivation for recursive builds:

- toplevel makefile
- integrating sources that have their own makefile

9.2.12 Customizing the Makefile

It is possible to add rules or otherwise customize the generated makefile by providing a `makefrag` file. When you run `opp_makemake`, it will automatically insert `makefrag` into the resulting `Makefile`. With the `-i` option, you can also name other files to be included into the `Makefile`.

`makefrag` will be inserted after the definitions but before the first rule, so it is possible to override existing definitions and add new ones, and also to override the default target.

`makefrag` can be useful if some of your source files are generated from other files (for example, you use generated NED files), or you need additional targets in your makefile or just simply want to override the default target in the makefile.

9.2.13 Projects with Multiple Source Trees

In the case of a large project, your source files may be spread across several directories and your project may generate more than one executable file (i.e. several shared libraries, examples etc.).

Once you have created your makefiles with `opp_makemake` in every source directory tree, you will need a toplevel makefile. The toplevel makefile usually calls only the makefiles recursively in the source directory trees.

9.2.14 A Multi-Directory Example

For a complex example of using `opp_makemake`, we will show how to create the makefiles for the Mobility Framework. First, take a look at the project's directory structure and find the directories that should be used as source trees:

```
mobility-framework
    bitmaps
    contrib <-- source tree (build libmfcontrib.so from this dir)
    core <-- source tree (build libmfcore.so from this dir)
    docs
    network
    template
    testSuite <-- source tree (build testSuite executable from this dir)
```

Additionally, there are dependencies between these output files: `mfcontrib` requires `mfcore` and `testSuite` requires `mfcontrib` (and indirectly `mfcore`).

First, we create the makefile for the core directory. The makefile will build a shared lib from all .cc files in the `core` subtree, and will name it `mfcore`):

```
$ cd core && opp_makemake -f --deep --make-so -o mfcore -O out
```

The `contrib` directory depends on `mfcore` so we use the `-L` and `-l` options to specify the library we should link with. Note that we must also add the include directories manually from the `core` source tree, because autodiscovery works only in the same source tree:

```
$ cd contrib && opp_makemake -f --deep --make-so -o mfcontrib -O out \\
  -I../core/basicModules -I../core/utils -L../out/$(CONFIGNAME)/core -lmfcore
```

The `testSuite` will be created as an executable file which depends on both `mfcontrib` and `mfcore`.

```
$ cd testSuite && opp_makemake -f --deep -o testSuite -O out
    -I../core/utils -I../core/basicModules -I../contrib/utils \\
    -I../contrib/applLayer -L../out/$(CONFIGNAME)/contrib -lmfcontrib
```

Now let us specify the dependencies between the above directories. Add the lines below to the `makefrag` file in the project directory root.

```
contrib_dir: core_dir
testSuite_dir: contrib_dir
```

Now the last step is to create a top-level makefile in the root of the project that calls the previously created makefiles in the correct order. We will use the `--nolink` option, exclude every subdirectory from the build (`-X.`), and explicitly call the above makefiles using `-d dirname`. `opp_makemake` will automatically include the above created `makefrag` file.

```
$ opp_makemake -f --nolink -O out -d testSuite -d core -d contrib -X.
```

Chapter 10

Configuring Simulations

10.1 The Configuration File

Configuration and input data for the simulation are in a configuration file usually called `omnetpp.ini`.

10.1.1 An Example

For a start, let us see a simple `omnetpp.ini` file which can be used to run the Fifo example simulation.

```
[General]
network = FifoNet
sim-time-limit = 100h
cpu-time-limit = 300s
#debug-on-errors = true
#record-eventlog = true

[Config Fifo1]
description = "low job arrival rate"
**.gen.sendIaTime = exponential(0.2s)
**.gen.msgLength = 100b
**.fifo.bitsPerSec = 1000bps

[Config Fifo2]
description = "high job arrival rate"
**.gen.sendIaTime = exponential(0.01s)
**.gen.msgLength = 10b
**.fifo.bitsPerSec = 1000bps
```

The file is grouped into *sections* named `[General]`, `[Config Fifo1]` and `[Config Fifo2]`, each one containing several *entries*.

10.1.2 File Syntax

An OMNeT++ configuration file is an ASCII text file, but non-ASCII characters are permitted in comments and string literals. This allows for using encodings that are a superset of ASCII, for example ISO 8859-1 and UTF-8. There is no limit on the file size or on the line length.

Comments may be placed at the end of any line after a hash mark, "#". Comments extend to the end of the line, and are ignored during processing. Blank lines are also allowed and ignored.

The file is line oriented, and consists of *section heading lines*, *key-value lines*, and *directive lines*:

1. *Section heading lines* contain a section name enclosed in square brackets.

2. *Key-value lines* have the *<key>=<value>* syntax; spaces are allowed (but not required) on both sides of the equal sign. If a line contains more than one equal sign, the leftmost one is taken as the key-value separator.

3. Currently there is only one kind of directive line, *include*. An include line starts with the `include` word, followed by the name of the file to be included.

Key-value lines may not occur above the first section heading line (except in included files, see later).

Keys may be further classified based on syntax alone:

1. Keys that do not contain dots represent global or per-run *configuration options*.

2. If a key contains a dot, its last component (substring after the last dot) is considered. If the last component contains a hyphen or is equal to `typename`, the key represents a *per-object configuration option*.

3. Otherwise, the key represents a *parameter assignment*. Thus, parameter assignment keys contain a dot, and no hyphen after the last dot.

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.

An example:

```
# This is a comment line
[General]                       # section heading
network = Foo                   # configuration option
debug-on-errors = false         # another configuration option

**.vector-recording = false     # per-object configuration option
**.app*.typename = "HttpClient" # per-object configuration option

**.app*.interval = 3s           # parameter value
**.app*.requestURL = "http://www.example.com/this-is-a-very-very-very-very\
-very-long-url?q=123456789"     # a two-line parameter value
```

10.1.3 File Inclusion

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 params1.ini
include params2.ini
include ../common/config.ini
...
```

You can also include files from other directories. If the included ini file further includes others, their path names will be understood as relative to the location of the file which contains the reference, rather than relative to the current working directory of the simulation.

This rule also applies to other file names occurring in ini files (such as the **`load-libs`**, **`output-vector-file`**, **`output-scalar-file`**, etc. options, and `xmldoc()` module parameter values.)

In included files, it is allowed to have key-value lines without first having a section heading line. File inclusion is conceptually handled as text substitution, except that a section heading in an included file will not change the current section the main file. The following example illustrates the rules:

```
# incl.ini
foo1 = 1          # no preceding section heading: these lines will go into
foo2 = 2          # whichever section the file is included into
[Config Bar]
bar = 3           # this will always go to into [Config Bar]
```

```
# omnetpp.ini
[General]
include incl.ini  # adds foo1/foo2 to [General], and defines [Config Bar] w/ bar
baz1 = 4          # include files don't change the current section, so these
baz2 = 4          # lines still belong to [General]
```

NOTE: The concept of file inclusion implies that include files may not make sense on their own. Thus, when you open an included ini file in an specialized ini file editor, the file contents may be flagged with errors and warnings. These errors/warnings disappear when the file is viewed as part of its main file.

10.2 Sections

An ini file may contain a `[General]` section and several `[Config <configname>]` sections. The order of the sections doesn't matter.

10.2.1 The [General] Section

The most commonly used options of the `[General]` section are the following.

- The **network** option selects the model to be set up and run.
- The length of the simulation can be set with the **sim-time-limit** and the **cpu-time-limit** options (the usual time units such as ms, s, m, h, etc. can be used).

Note that the NED files loaded by the simulation may contain several networks, and any of them may be specified in the **network** option.

10.2.2 Named Configurations

Named configurations are sections of the form `[Config <configname>]`, where *<configname>* is by convention a camel-case string that starts with a capital letter: `Config1`, `WirelessPing`, `OverloadedFifo`, etc. For example, `omnetpp.ini` for an Aloha simulation might have the following skeleton:

```
[General]
...
[Config PureAloha]
...
[Config SlottedAloha1]
...
[Config SlottedAloha2]
...
```

Some configuration options (such as user interface selection) are only accepted in the `[General]` section, but most of them can go into `Config` sections as well.

When you run a simulation, you need to select one of the configurations to be activated. In Cmdenv, this is done with the `-c` command-line option:

```
$ aloha -c PureAloha
```

The simulation will then use the contents of the `[Config PureAloha]` section to set up the simulation. (Tkenv, of course, lets you select the configuration from a dialog.)

10.2.3 Section Inheritance

Actually, when you activate the PureAloha configuration, the contents of the `[General]` section will also be taken into account: if some configuration option or parameter value is not found in `[Config PureAloha]`, then the search will continue in the `[General]` section. In other words, lookups in `[Config PureAloha]` will fall back to `[General]`. The `[General]` section itself is optional; when it is absent, it is treated like an empty `[General]` section.

All named configurations fall back to `[General]` by default. However, for each configuration it is possible to specify the fall-back section or a list of fallback sections explicitly, using the **extends** key. Consider the following ini file skeleton:

```
[General]
...
[Config SlottedAlohaBase]
...
[Config LowTrafficSettings]
...
[Config HighTrafficSettings]
```

```
...

[Config SlottedAloha1]
extends = SlottedAlohaBase, LowTrafficSettings
...
[Config SlottedAloha2]
extends = SlottedAlohaBase, HighTrafficSettings
...
[Config SlottedAloha2a]
extends = SlottedAloha2
...
[Config SlottedAloha2b]
extends = SlottedAloha2
...
```

If you activate the `SlottedAloha2b` configuration, lookups will consider sections in the following order (this is also called the *section fallback chain*): `SlottedAloha2b`, `SlottedAloha2`, `SlottedAlohaBase`, `HighTrafficSettings`, `General`.

The effect is the same as if the contents of the sections SlottedAloha2b, SlottedAloha2, SlottedAlohaBase, HighTrafficSettings and General were copied together into one section, one after another, `[Config SlottedAloha2b]` being at the top, and `[General]` at the bottom. Lookups always start at the top, and stop at the first matching entry.

The order of the sections in the *fallback chain* is computed using the *C3 linearization algorithm* ([BCH+96]):

The *fallback chain* of a configuration `A` is

- if `A` does not have an **`extends`** key then `A`, `General`
- otherwise the merge of the configurations enumerated in the **`extends`** key, and all of their *fallback section chains*. The merge is *monotonic*: if some configuration `X` precedes configuration `Y` in one of the input chains, it will precede it in the output chain too.

The *section fallback chain* can be printed by the `-X` option of the command line of the simulation program:

```
$ aloha -X SlottedAloha2b
OMNeT++ Discrete Event Simulation
...
Config SlottedAloha2b
Config SlottedAloha2
Config SlottedAlohaBase
Config HighTrafficSettings
General
```

The *section fallback* concept is similar to multiple inheritance in object-oriented languages, and benefits are similar too; you can factor out the common parts of several configurations into a "base" configuration, and additionally you can reuse existing configurations (as opposed to copying them) by using them as a base. In practice you will often have "abstract" configurations too (in the C++/Java sense), which assign only a subset of parameters and leave the others open, to be assigned in derived configurations.

If you are experimenting a lot with different parameter settings of a simulation model, these techniques will make it much easier to manage ini files.

10.3 Assigning Module Parameters

Simulations get input via module parameters, which can be assigned a value in NED files or in `omnetpp.ini` – in this order. Since parameters assigned in NED files cannot be overridden in `omnetpp.ini`, one can think about them as being "hardcoded". In contrast, it is easier and more flexible to maintain module parameter settings in `omnetpp.ini`.

In `omnetpp.ini`, module parameters are referred to by their full paths (hierarchical names). This name consists of the dot-separated list of the module names (from the top-level module down to the module containing the parameter), plus the parameter name (see section 7.1.5).

An example `omnetpp.ini` which sets the `numHosts` parameter of the toplevel module and the `transactionsPerSecond` parameter of the `server` module:

```
[General]
Network.numHosts = 15
Network.server.transactionsPerSecond = 100
```

Typename pattern assignments are also accepted:

```
[General]
Network.host[*].app.typename = "PingApp"
```

10.3.1 Using Wildcard Patterns

Models can have a large number of parameters to be configured, and it would be tedious to set them one-by-one in `omnetpp.ini`. OMNeT++ supports *wildcard patterns* which allow for setting several model parameters at once. The same pattern syntax is used for per-object configuration options; for example `<object-path-pattern>.record-scalar`, or `<module-path-pattern>.rng-<N>`.

The pattern syntax is a variation on Unix *glob*-style patterns. The most apparent differences to globbing rules are the distinction between `*` and `**`, and that character ranges should be written with curly braces instead of square brackets; that is, *any-letter* is expressed as `{a-zA-Z}` and not as `[a-zA-Z]`, because square brackets are reserved for the notation of module vector indices.

Pattern syntax:

- `?` : matches any character except dot (.)
- `*` : matches zero or more characters except dot (.)
- `**` : matches zero or more characters (any character)
- `{a-f}` : *set*: matches a character in the range a-f
- `{^a-f}`: *negated set*: matches a character NOT in the range a-f
- `{38..150}` : *numeric range*: any number (i.e. sequence of digits) in the range 38..150, inclusive; both limits are optional
- `[38..150]` : *index range*: any number in square brackets in the range 38..150, inclusive; both limits are optional
- backslash (\) : takes away the special meaning of the subsequent character

Precedence

If you use wildcards, the order of entries is important; if a parameter name matches several wildcard-patterns, the *first* matching occurrence is used. This means that you need to list specific settings first, and more general ones later. Catch-all settings should come last.

An example ini file:

```
[General]
*.host[0].waitTime = 5ms    # specifics come first
*.host[3].waitTime = 6ms
*.host[*].waitTime = 10ms   # catch-all comes last
```

Asterisk vs Double Asterisk

The `*` wildcard is for matching a single module or parameter name in the path name, while `**` can be used to match several components in the path. For example, `**.queue*.bufSize` matches the `bufSize` parameter of any module whose name begins with `queue` in the model, while `*.queue*.bufSize` or `net.queue*.bufSize` selects only queues immediately on network level. Also note that `**.queue**.bufSize` would match `net.queue1.foo.bar.bufSize` as well!

Sets, Negated Sets

Sets and negated sets can contain several character ranges and also enumeration of characters. For example, `{_a-zA-Z0-9}` matches any letter or digit, plus the underscore; `{xyzc-f}` matches any of the characters x, y, z, c, d, e, f. To include '-' in the set, put it at a position where it cannot be interpreted as character range, for example: `{a-z-}` or `{-a-z}`. If you want to include '}' in the set, it must be the first character: `{}a-z}`, or as a negated set: `{^}a-z}`. A backslash is always taken as a literal backslash (and not as an escape character) within set definitions.

Numeric Ranges and Index Ranges

Only nonnegative integers can be matched. The start or the end of the range (or both) can be omitted: `{10..}`, `{..99}` or `{..}` are valid numeric ranges (the last one matches any number). The specification must use exactly two dots. Caveat: `*{17..19}` will match `a17`, `117` and `963217` as well, because the `*` can also match digits!

An example for numeric ranges:

```
[General]
*.*.queue[3..5].bufSize = 10
*.*.queue[12..].bufSize = 18
*.*.queue[*].bufSize = 6  # this will only affect queues 0,1,2 and 6..11
```

10.3.2 Using the Default Values

It is also possible to utilize the default values specified in the NED files. The <*parameter-fullpath*>`=default` setting assigns the default value to a parameter if it has one.

The <*parameter-fullpath*>=ask setting will try to get the parameter value interactively from the user.

If a parameter was not set but has a default value, that value will be assigned. This is like having a **=default line at the bottom of the [General] section.

If a parameter was not set and has no default value, that will either cause an error or will be interactively prompted for, depending on the particular user interface.

> **NOTE:** In Cmdenv you must explicitly enable the interactive mode with the --cmdenv-interactive=true option otherwise you will get an error when running the simulation.

More precisely, parameter resolution takes place as follows:

1. If the parameter is assigned in NED, it cannot be overridden in the configuration. The value is applied and the process finishes.
2. If the first match is a value line (matches <*parameter-fullpath*>=<*value*>), the value is applied and the process finishes.
3. If the first match is a <*parameter-fullpath*>=default line, the default value is applied and the process finishes.
4. If the first match is a <*parameter-fullpath*>=ask line, the parameter will be asked from the user interactively (UI dependent).
5. If there was no match and the parameter has a default value, it is applied and the process finishes.
6. Otherwise the parameter is declared unassigned, and handled accordingly by the user interface. It may be reported as an error, or may be asked from the user interactively.

10.4 Parameter Studies

It is quite common in simulation studies that the simulation model is run several times with different parameter settings, and the results are analyzed in relation to the input parameters. OMNeT++ 3.x had no direct support for batch runs, and users had to resort to writing shell (or Python, Ruby, etc.) scripts that iterated over the required parameter space, to generate a (partial) ini file and run the simulation program in each iteration.

OMNeT++ 4.x largely automates this process, and eliminates the need for writing batch execution scripts. It is the ini file where the user can specify iterations over various parameter settings. Here is an example:

```
[Config AlohaStudy]
*.numHosts = ${1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential(${0.2, 0.4, 0.6}s)
```

This parameter study expands to 8*3 = 24 simulation runs, where the number of hosts iterates over the numbers 1, 2, 5, 10, 20, 30, 40, 50, and for each host count three simulation runs will be done, with the generation interval being exponential(0.2), exponential(0.4), and exponential(0.6).

How does it work? First of all, Cmdenv with the -x option will tell you how many simulation runs a given section expands to. (You will of course use Cmdenv for batch runs, not Tkenv.)

```
$ aloha -u Cmdenv -x AlohaStudy

OMNeT++ Discrete Event Simulation
...
Config: AlohaStudy
Number of runs: 24
```

If you add the `-g` option, the program will also print out the values of the iteration variables for each run. (Use `-G` for even more info.) Note that the parameter study actually maps to nested loops, with the last `${...}` becoming the innermost loop. The iteration variables are just named `$0` and `$1` – we'll see that it is possible to give meaningful names to them. Please ignore the `$repetition=0` part in the printout for now.

```
$ aloha -u Cmdenv -x AlohaStudy -g
OMNeT++ Discrete Event Simulation
...
Config: AlohaStudy
Number of runs: 24
Run 0: $0=1, $1=0.2, $repetition=0
Run 1: $0=1, $1=0.4, $repetition=0
Run 2: $0=1, $1=0.6, $repetition=0
Run 3: $0=2, $1=0.2, $repetition=0
Run 4: $0=2, $1=0.4, $repetition=0
Run 5: $0=2, $1=0.6, $repetition=0
Run 6: $0=5, $1=0.2, $repetition=0
Run 7: $0=5, $1=0.4, $repetition=0
...
Run 19: $0=40, $1=0.4, $repetition=0
Run 20: $0=40, $1=0.6, $repetition=0
Run 21: $0=50, $1=0.2, $repetition=0
Run 22: $0=50, $1=0.4, $repetition=0
Run 23: $0=50, $1=0.6, $repetition=0
```

Any of these runs can be executed by passing the `-r <runnumber>` option to Cmdenv. So, the task is now to run the simulation program 24 times, with `-r` running from 0 through 23:

```
$ aloha -u Cmdenv -c AlohaStudy -r 0
$ aloha -u Cmdenv -c AlohaStudy -r 1
$ aloha -u Cmdenv -c AlohaStudy -r 2
...
$ aloha -u Cmdenv -c AlohaStudy -r 23
```

This batch can be executed either from the OMNeT++ IDE (where you are prompted to pick an executable and an ini file, choose the configuration from a list, and just click Run), or using a little command-line batch execution tool (`opp_runall`) supplied with OMNeT++.

Actually, it is also possible to get Cmdenv execute all runs in one go, by simply omitting the `-r` option.

```
$ aloha -u Cmdenv -c AlohaStudy

OMNeT++ Discrete Event Simulation
Preparing for running configuration AlohaStudy, run #0...
...
Preparing for running configuration AlohaStudy, run #1...
```

```
...
...
Preparing for running configuration AlohaStudy, run #23...
```

However, this approach is not recommended, because it is more susceptible to C++ programming errors in the model. (For example, if any of the runs crashes, the whole batch is terminated – which may not be what the user wants.)

10.4.1 Iterations

Let us have a look at the example ini file in the previous section again:

```
[Config AlohaStudy]
*.numHosts = ${1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential( ${0.2, 0.4, 0.6}s )
```

The `${...}` syntax specifies an iteration. It is sort of a macro: at each run, the whole `${...}` string is textually replaced with the current iteration value. The values to iterate over do not need to be numbers (unless you want to use the *"a..b"* or *"a..b step c"* syntax), and the substitution takes place even inside string constants. So, the following examples are all valid (note that textual substitution is used):

```
*.param = 1 + ${1e-6, 1/3, sin(0.5)}
    ==> *.param = 1 + 1e-6
        *.param = 1 + 1/3
        *.param = 1 + sin(0.5)
*.greeting = "We will simulate ${1,2,5} host(s)."
    ==> *.greeting = "We will simulate 1 host(s)."
        *.greeting = "We will simulate 2 host(s)."
        *.greeting = "We will simulate 5 host(s)."
```

To write a literal `${..}` inside a string constant, quote the left brace with a backslash: `$\{..}`.

NOTE: Inside `${..}`, the values are separated with commas. However, not every comma is taken as a value separator because the parser tries to be smart about what you meant. Commas inside (nested) parentheses, brackets or curly braces are ignored so that `${uniform(0,3)}` is parsed as one value and not as `uniform(0` plus `3)`. Commas, curly braces and other charachers inside double-quoted string literals are also ignored, so `${"Hello, world"}` yields a single `"Hello, world"` string and not `"Hello` plus `world"`. It is assumed that string literals use backslash as an escape characher, like in C/C++ and NED. If you want to have a literal comma or close-brace inside a value, you need to escape it with a backslash: `${foo\,bar\}baz}` will parse as a single value, `foo,bar}baz`. Backslashes themselves must be doubled. As the above examples illustrate, the parser removes one level of backslashes, except inside string literals where they are left intact.

10.4.2 Named Iteration Variables

You can assign names to iteration variables, which has the advantage that you will see meaningful names instead of `$0` and `$1` in the Cmdenv output, and also lets you refer to the variables at more than one place in the ini file. The syntax is `${<varname>=<iteration>}`, and variables can be referred to simply as `${<varname>}`:

```
[Config Aloha]
*.numHosts = ${N=1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential( ${mean=0.2, 0.4, 0.6}s )
**.greeting = "There are ${N} hosts"
```

The scope of the variable name is the section that defines it, plus sections based on that section (via **extends**).

Referencing Other Iteration Variables

Iterations may refer to other iteration variables, using the dollar syntax (`$var`) or the dollar-brace syntax (`${var}`).

This feature makes it possible to have loops where the inner iteration range depends on the outer one. An example:

```
**.foo = ${i=1..10}  # outer loop
**.bar = ${j=1..$i}  # inner loop depends on $i
```

When needed, the default top-down nesting order of iteration loops is modified (loops are reordered) to ensure that expressions only refer to more outer loop variables, but not to inner ones. When this is not possible, an error is generated with the "circular dependency" message.

For instance, in the following example the loops will be nested in *k* - *i* - *j* order, *k* being the outermost and *j* the innermost loop:

```
**.foo = ${i=0..$k}   # must be inner to $k
**.bar = ${j=$i..$k}  # must be inner to both $i and $k
**.baz = ${k=1..10}   # may be the outermost loop
```

And the next example will stop with an error because there is no "good" ordering:

```
**.foo = ${i=0..$j}
**.bar = ${j=0..$k}
**.baz = ${k=0..$i} # --> error: circular references
```

Variables are substituted *textually*, and the result is normally *not* evaluated as an arithmetic expression. The result of the substitution is only evaluated where needed, namely in the three arguments of iteration ranges (*from*, *to*, *step*), and in the value of the **constraint** configuration option.

To illustrate textual substitution, consider the following contorted example:

```
**.foo = ${i=1..3, 1s+, -}001s
```

Here, the `foo` NED parameter will receiving the following values in subsequent runs: `1001s`, `2001s`, `3001s`, `1s+001s`, `-001s`.

CAUTION: Due to textual substitution, variables in arithmetic expressions should be protected with parentheses – just like in C/C++ function-style macros. Consider the following example:

```
**.foo = ${i=10}
**.bar = ${j=$i+5}
**.baz = ${k=2*$j}    # bogus! $j should be written as ($j)
constraint = $i+50 < 2*$j  # ditto: should use ($i) and ($j)
```

Here, the `baz` parameter will receive the string `2*10+5` after the substitutions and hence evaluate to 25 instead of the correct $2*(10+5)=30$; the constraint expression is similarly wrong. Mind the parens!

Substitution also works inside string constants within iterations (`${..}`).

```
**.foo = "${i=Jo}"  # -> Jo
**.bar = ${"Hi $i", "Hi ${i}hn"}  # -> Hi Jo /John
```

However, outside iterations the plain dollar syntax is not understood, only the dollar-brace syntax is:

```
**.foo = "${i=Day}"
**.baz = "Good $i"      # -> remains "Good $i"
**.baz = "Good ${i}"    # -> becomes "Good Day"
```

Rationale: The text substitution model was chosen for greater flexibility as well as the ability to produce more consistent semantics. The advantages outweigh the inconvenience of having to parenthesize variable references in arithmetic expressions.

10.4.3 Parallel Iteration

The body of an iteration may end in an exclamation mark followed by the name of another iteration variable. This syntax denotes a *parallel iteration*. A parallel iteration does not define a loop of its own, but rather, the sequence is advanced in lockstep with the variable after the "!". In other words, the "!" syntax chooses the *kth* value from the iteration, where *k* is the position (iteration count) of the iteration variable after the "!".

An example:

```
**.plan =     ${plan= "A", "B", "C", "D"}
**.numHosts = ${hosts= 10,  20,  50, 100 ! plan}
**.load =     ${load= 0.2, 0.3, 0.3, 0.4 ! plan}
```

In the above example, the only loop is defined by the first line, the `plan` variable. The other two iterations, `hosts` and `load` just follow it; for the first value of `plan` the first values of `host` and `load` are selected, and so on.

10.4.4 Predefined Variables, Run ID

There are a number of predefined variables: `${configname}` and `${runnumber}` with the obvious meanings; `${network}` is the name of the network that is simulated; `${processid}` and `${datetime}` expand to the OS process id of the simulation and the time it was started; and there are some more: `${runid}`, `${iterationvars}` and `${repetition}`.

`${runid}` holds the *run ID*. When a simulation is run, a a run ID is generated that uniquely identifies that instance of the simulation: if you run the same thing again, it will get a different run ID. Run ID is a concatenation of several variables like `${configname}`, `${runnumber}`, `${datetime}` and `${processid}`. This yields an identifier that is unique "enough" for all practical purposes, yet it is meaningful for humans. The run ID is recorded into result files written during the simulation, and can be used to match vectors and scalars written by the same simulation run.

10.4.5 Constraint Expression

In cases when not all combinations of the iteration variables make sense or need to be simulated, it is possible to specify an additional constraint expression. This expression is interpreted as a conditional (an "if" statement) within the innermost loop, and it must evaluate to `true` for the variable combination to generate a run. The expression should be given with the **`constraint`** configuration option. An example:

```
*.numNodes = ${n=10..100 step 10}
**.numNeighbors = ${m=2..10 step 2}
constraint = ($m) <= sqrt($n)  # note: parens needed due to textual substitution
```

The expression syntax supports most C language operators including boolean, conditional and binary shift operations, and most `<math.h>` functions; data types are boolean, double and string. The expression must evaluate to a boolean.

> **NOTE:** Remember that variables are substituted textually into the expresssion, so they must be protected with parentheses to preserve evaluation order.

10.4.6 Repeating Runs with Different Seeds

It is directly supported to perform several runs with the same parameters but different random number seeds. There are two configuration options related to this: **`repeat`** and **`seed-set`**. The first one simply specifies how many times a run needs to be repeated. For example,

```
repeat = 10
```

causes every combination of iteration variables to be repeated 10 times, and the `${repetition}` predefined variable holds the loop counter. Indeed, `repeat=10` is equivalent to adding `${repetition=0..9}` to the ini file. The `${repetition}` loop always becomes the innermost loop.

The **`seed-set`** configuration key affects seed selection. Every simulation uses one or more random number generators (as configured by the **`num-rngs`** key), for which the simulation kernel can automatically generate seeds. The first simulation run may use one set of seeds (seed set 0), the second run may use a second set (seed set 1), and so on. Each set contains as many seeds as there are RNGs configured. All automatic seeds generate random number sequences that are far apart in the RNG's cycle, so they will never overlap during simulations.

> **NOTE:** Mersenne Twister, the default RNG of OMNeT++ has a cycle length of 2^{19937}, which is more than enough for any conceivable purpose.

The **`seed-set`** key tells the simulation kernel which seed set to use. It can be set to a concrete number (such as `seed-set=0`), but it usually does not make sense as it would cause

every simulation to run with exactly the same seeds. It is more practical to set it to either `${runnumber}` or to `${repetition}`. The default setting is `${runnumber}`:

```
seed-set = ${runnumber}    # this is the default
```

This causes every simulation run to execute with a unique seed set. The second option is:

```
seed-set = ${repetition}
```

where all `$repetition=0` runs will use the same seeds (seed set 0), all `$repetition=1` runs use another seed set, `$repetition=2` a third seed set, etc.

To perform runs with manually selected seed sets, you can just define an iteration for the **`seed-set`** key:

```
seed-set = ${5,6,8..11}
```

In this case, the **`repeat`** key should be left out, as **`seed-set`** already defines an iteration and there is no need for an extra loop.

It is of course also possible to manually specify individual seeds for simulations. The parallel iteration feature is very convenient here:

```
repeat = 4
seed-1-mt = ${53542, 45732, 47853, 33434 ! repetition}
seed-2-mt = ${75335, 35463, 24674, 56673 ! repetition}
seed-3-mt = ${34542, 67563, 96433, 23567 ! repetition}
```

The meaning of the above is this: in the first repetition, the first column of seeds is chosen, for the second repetition, the second column, etc. The "!" syntax chooses the *kth* value from the iteration, where *k* is the position (iteration count) of the iteration variable after the "!". Thus, the above example is equivalent to the following:

```
# no repeat= line!
seed-1-mt = ${seed1 = 53542, 45732, 47853, 33434}
seed-2-mt = ${        75335, 35463, 24674, 56673 ! seed1}
seed-3-mt = ${        34542, 67563, 96433, 23567 ! seed1}
```

That is, the iterators of `seed-2-mt` and `seed-3-mt` are advanced in lockstep with the `seed1` iteration.

10.4.7 Experiment-Measurement-Replication

We have introduced three concepts that are useful for organizing simulation results generated by batch executions or several batches of executions.

During a simulation study, a user prepares several *experiments*. The purpose of an experiment is to find out the answer to questions like *"how does the number of nodes affect response times in the network?"* For an experiment, several *measurements* are performed on the simulation model, and each measurement runs the simulation model with a different set of parameters. To eliminate the bias introduced by the particular random number stream used for the simulation, several *replications* of every measurement are run with different random number seeds, and the results are averaged.

OMNeT++ result analysis tools can take advantage of the *experiment*, *measurement* and *replication* labels recorded into result files, and display simulation runs and recorded results accordingly on the user interface.

These labels can be explicitly specified in the ini file using the **experiment-label**, **measurement-label** and **replication-label** config options. If they are missing, the default is the following:

```
experiment-label = "${configname}"
measurement-label = "${iterationvars}"
replication-label = "#${repetition},seed-set=<seedset>"
```

That is, the default experiment label is the configuration name; the measurement label is concatenated from the iteration variables; and the replication label contains the repeat loop variable and seed-set. Thus, for our first example the *experiment-measurement-replication* tree would look like this:

```
"PureAloha"--experiment
  $N=1,$mean=0.2 -- measurement
    #0, seed-set=0 -- replication
    #1, seed-set=1
    #2, seed-set=2
    #3, seed-set=3
    #4, seed-set=4
  $N=1,$mean=0.4
    #0, seed-set=5
    #1, seed-set=6
    ...
    #4, seed-set=9
  $N=1,$mean=0.6
    #0, seed-set=10
    #1, seed-set=11
    ...
    #4, seed-set=14
  $N=2,$mean=0.2
    ...
  $N=2,$mean=0.4
    ...
    ...
```

The *experiment-measurement-replication* labels should be enough to reproduce the same simulation results, given of course that the ini files and the model (NED files and C++ code) haven't changed.

Every instance of running the simulation gets a unique run ID. We can illustrate this by listing the corresponding run IDs under each repetition in the tree. For example:

```
"PureAloha"
  $N=1,$mean=0.2
    #0, seed-set=0
      PureAloha-0-20070704-11:38:21-3241
      PureAloha-0-20070704-11:53:47-3884
      PureAloha-0-20070704-16:50:44-4612
    #1, seed-set=1
      PureAloha-1-20070704-16:50:55-4613
    #2, seed-set=2
      PureAloha-2-20070704-11:55:23-3892
```

```
PureAloha-2-20070704-16:51:17-4615
...
```

The tree shows that ("PureAloha", "$N=1,$mean=0.2", "#0, seed-set=0") was run three times. The results produced by these three executions should be identical, unless, for example, some parameter was modified in the ini file, or a bug got fixed in the C++ code.

We believe that the default way of generating *experiment-measurement-replication* labels is useful and sufficient for the majority of simulation studies. However, you can customize it if needed. For example, here is a way to join two configurations into one experiment:

```
[Config PureAloha_Part1]
experiment-label = "PureAloha"
...
[Config PureAloha_Part2]
experiment-label = "PureAloha"
...
```

Measurement and replication labels can be customized in a similar way, making use of named iteration variables, `${repetition}`, `${runnumber}` and other predefined variables. One possible benefit is to customize the generated measurement and replication labels. For example:

```
[Config PureAloha_Part1]
measurement = "${N} hosts, exponential(${mean}) packet generation interval"
```

One should be careful with the above technique though, because if some iteration variables are left out of the measurement labels, runs with all values of those variables will be grouped together to the same replications.

10.5 Configuring the Random Number Generators

The random number architecture of OMNeT++ was already outlined in section 7.4. Here we'll cover the configuration of RNGs in `omnetpp.ini`.

10.5.1 Number of RNGs

The **`num-rngs`** configuration option sets the number of random number generator instances (i.e. random number streams) available for the simulation model (see 7.4). Referencing an RNG number greater or equal to this number (from a simple module or NED file) will cause a runtime error.

10.5.2 RNG Choice

The **`rng-class`** configuration option sets the random number generator class to be used. It defaults to `"cMersenneTwister"`, the Mersenne Twister RNG. Other available classes are `"cLCG32"` (the "legacy" RNG of OMNeT++ 2.3 and earlier versions, with a cycle length of $2^{31}-2$), and `"cAkaroaRNG"` (Akaroa's random number generator, see section 11.5).

10.5.3 RNG Mapping

The RNG numbers used in simple modules may be arbitrarily mapped to the actual random number streams (actual RNG instances) from `omnetpp.ini`. The mapping allows for great flexibility in RNG usage and random number streams configuration – even for simulation models which were not written with RNG awareness.

RNG mapping may be specified in `omnetpp.ini`. The syntax of configuration entries is the following.

```
[General]
<modulepath>.rng-N = M  # where N,M are numeric, M < num-rngs
```

This maps module-local RNG N to physical RNG M. The following example maps all `gen` module's default (N=0) RNG to physical RNG 1, and all `noisychannel` module's default (N=0) RNG to physical RNG 2.

```
[General]
num-rngs = 3
**.gen[*].rng-0 = 1
**.noisychannel[*].rng-0 = 2
```

This mapping allows variance reduction techniques to be applied to OMNeT++ models, without any model change or recompilation.

10.5.4 Automatic Seed Selection

Automatic seed selection is used for an RNG if you do not explicitly specify seeds in `omnetpp.ini`. Automatic and manual seed selection can co-exist; for a particular simulation, some RNGs can be configured manually, and some automatically.

The automatic seed selection mechanism uses two inputs: the *run number* and the *RNG number*. For the same run number and RNG number, OMNeT++ always selects the same seed value for any simulation model. If the run number or the RNG number is different, OMNeT++ does its best to choose different seeds which are also sufficiently separated in the RNG's sequence so that the generated sequences don't overlap.

The run number can be specified either in in `omnetpp.ini` (e.g. via the **`cmdenv-runs-to-execute`** option) or on the command line:

```
./mysim -r 1
./mysim -r 2
./mysim -r 3
```

For the `cMersenneTwister` random number generator, selecting seeds so that the generated sequences don't overlap is easy, due to the extremely long sequence of the RNG. The RNG is initialized from the 32-bit seed value $seed = runNumber * numRngs + rngNumber$. (This implies that simulation runs participating in the study should have the same number of RNGs set). [1]

For the `cLCG32` random number generator, the situation is more difficult, because the range of this RNG is rather short ($2^{31} - 1$, about 2 billion). For this RNG, OMNeT++ uses a table of 256 pre-generated seeds, equally spaced in the RNG's sequence. Index into the table is calculated with the $runNumber * numRngs + rngNumber$ formula. Care should be taken that

[1] While (to our knowledge) no one has proven that the seeds 0,1,2,... are well apart in the sequence, this is probably true, due to the extremely long sequence of MT. The author would however be interested in papers published about seed selection for MT.

one doesn't exceed 256 with the index, or it will wrap and the same seeds will be used again. It is best not to use the `cLCG32` at all – `cMersenneTwister` is superior in every respect.

10.5.5 Manual Seed Configuration

In some cases you may want to manually configure seed values. Reasons for doing that may be that you want to use variance reduction techniques, or you may want to use the same seeds for several simulation runs.

To manually set seeds for the Mersenne Twister RNG, use the `seed-`*k*`-mt` option, where *k* is the RNG index. An example:

```
[General]
num-rngs = 3
seed-0-mt = 12
seed-1-mt = 9
seed-2-mt = 7
```

For the now obsolete cLCG32 RNG, the name of the corresponding option is `seed-`*k*`-lcg32`, and OMNeT++ provides a standalone program called `opp_lcg32_seedtool` to generate good seed values that are sufficiently separated in the RNG's sequence.

Chapter 11

Running Simulations

11.1 Introduction

This chapter describes how to run simulations. It covers basic usage, user interfaces, batch runs, how to use Akaroa, and also explains how to solve the most common errors.

11.1.1 Running a Simulation Executable

By default, the output of an `opp_makemake`-generated makefile is a simulation executable that can be run directly. In simple cases, this executable can be run without command-line arguments, but usually one will need to specify options to specify what ini file to use, which user interface to activate, where to load NED files from, and so on.

Getting Help

The following sections describe the most frequently used command-line options. To get a complete list of supported command line options, run the `opp_run` command (or any other simulation executable) with the `-h` option.

```
$ opp_run -h
```

Specifying Ini Files

The default ini file is `omnetpp.ini`, and is loaded if no other ini file is given on the command line.

Ini files can be specified both as plain arguments and with the `-f` option, so the following two commands are equivalent:

```
$ ./fifo experiment.ini common.ini
$ ./fifo -f experiment.ini -f common.ini
```

Multiple ini files can be given, and their contents will be merged. This allows for partitioning the configuration into separate files, for example to simulation options, module parameters and result recording options.

Specifying the NED Path

NED files are loaded from directories listed on the NED path. More precisely, they are loaded from the listed directories and their whole subdirectory trees. Directories are separated with a semicolon (`;`).

> **NOTE:** Semicolon is used as separator on both Unix and Windows.

The NED path can be specified in several ways:

- using the `NEDPATH` environment variable
- using the `-n` command-line option
- in ini files, with the **`ned-path`** configuration option

NED path resolution rules are as follows:

- OMNeT++ checks for NED path specified on the command line with the `-n` option
- if not found on the command line, it checks for the NEDPATH environment variable
- the **`ned-path`** option value from the ini file is appended to the result of the above steps
- if the result is still empty, it falls back to "." (the current directory)

Selecting a User Interface

OMNeT++ simulations can be run under different user interfaces. Currently the following user interfaces are supported:

- Tkenv: Tcl/Tk-based graphical, windowing user interface
- Cmdenv: command-line user interface for batch execution

You would typically test and debug your simulation under Tkenv, then run actual simulation experiments from the command line or shell script, using Cmdenv. Tkenv is also better suited for educational or demonstration purposes.

Both Tkenv and Cmdenv are provided in the form of a library, and you may choose between them by linking one or both into your simulation executable. (Creating the executable was described in chapter 9). Both user interfaces are supported on Unix and Windows platforms.

You can choose which runtime environment is included in your simulation executable when you generate your makefile. By default, both Tkenv and Cmdenv is linked in so you can choose between them during runtime, but it is possible to specify only a single user interface with the `-u Cmdenv` or `-u Tkenv` option on the `opp_makemake` command line. This can be useful if you intend to run your simulations on a machine where Tcl/Tk is not installed.

By default, Tkenv will be used if both runtime environments are present in your executable. You explicitly select a user interface by adding the `user-interface=Cmdenv` (or `=Tkenv`) option in your ini file, or by specifying `-u Cmdenv` or `-u Tkenv` on the command line. If both the config option and the command line option are present, the command line option takes precedence.

Selecting a Configuration and Run Number

Configurations can be selected with the `-c <configname>` command line option. If you do not specify the configuration and you are running under:

- Tkenv: the runtime environment will prompt you to choose one.
- Cmdenv: the `General` configuration will be executed.

User interfaces may support the `-r runnumber` option to select runs, either one or more, depending on the type of the user interface.

There are several command line options to get information about the iteration variables and the number of runs in the configurations:

- `-a` – Prints all configuration names and the number of runs in them.
- `-x <configname>` – Prints the number of runs available in the given configuration.
- `-g` – Prints the unrolled configuration (together with the -x option) and expands the iteration variables.
- `-G` – Prints even more details than -g.

Loading Extra Libraries

OMNeT++ allows you to load shared libraries at runtime. This means that you can create simulation models as a shared library and load the model later into a different executable without the need to explicitly link against that library. This approach has several advantages.

- It is possible to distribute the model as a shared library. Others may be able to use it without recompiling it.
- You can split a large model into smaller, reusable components.
- You can mix several models (even from different projects) without the need of linking or compiling.

Use the `-l libraryname` command line option to load a library dynamically. OMNeT++ will attempt to load it using the `dlopen()` or `LoadLibrary()` functions and automatically registers all simple modules in the library.

The prefix and suffix from the library name can be omitted (the extensions `.dll`, `.so`, `.dylib`, and also the common `lib` prefix on Unix systems). This means that you can specify the library name in a platform independent way: if you specify `-l foo`, then OMNeT++ will look for `foo.dll`, `libfoo.dll`, `libfoo.so` or `libfoo.dylib`, depending on the platform.

It is also possible to specify the libraries to be loaded in the ini file with the **`load-libs`** configuration option. The values from the command line and the config file will be merged.

> **NOTE:** Runtime loading is not needed if your executable or shared lib was already linked against the library in question. In that case, the platform's dynamic loader will automatically load the library.

NOTE: You must ensure that the library can be accessed by OMNeT++. Either specify the library name with a full path (pre- and postfixes of the library file name still can be omitted), or adjust the shared library path environment variable of your OS (`PATH` on Windows, `LD_LIBRARY_PATH` on Unix, and `DYLD_LIBRARY_PATH` on Mac OS X.)

11.1.2 Running a Shared Library

Shared libraries can be run using the `opp_run` program. Both `opp_run` and simulation executables are capable of loading additional shared libraries; actually, `opp_run` is nothing else than an empty simulation executable.

Example:

```
opp_run -l mymodel
```

The above example will load the model found in `libmymodel.so` and execute it.

11.1.3 Controlling the Run

There are several useful configuration options that control how a simulation is run.

- **`cmdenv-express-mode`** – Provides only minimal status updates on the console.
- **`cmdenv-interactive`** – Allows the simulation to ask missing parameter values interactively
- **`cmdenv-status-frequency`** – Controls how often the status is written to the console.
- **`cpu-time-limit`** – Limits how long the simulation should run (in wall clock time)
- **`sim-time-limit`** – Limits how long the simulation should run (in simulation time)
- **`debug-on-errors`** – If the runtime detects any error, it will generate a breakpoint so you will be able to check the location and the context of the problem in your debugger.
- **`fingerprint`** – The simulation kernel computes a checksum while running the simulation. It is calculated from the module id and from the current simulation time of each event. If you specify the **`fingerprint`** option in the config file, the simulation runtime will compare the computed checksum with the provided one. If there is a difference it will generate an error. This feature is very useful if you make some cosmetic changes to your source and want to be reasonable sure that your changes did not alter the behaviour of the model.

  WARNING: The value of the calculated fingerprint is heavily dependent on the accuracy of the floating point arithmetic. There are differences between the floating point handling of AMD and Intel CPUs. Running under a processor emulator software like `valgrind` may also produce a different fingerprint. This is normal. Hint: see gcc options `-mfpmath=sse -msse2`.

- **`record-eventlog`** – You can turn on the recording of the simulator events. The resulting file can be analyzed later in the IDE with the sequence chart tool.

NOTE: It is also possible to specify a configuration option on the command line (in which case the command line takes precedence). To do so, prefix the option name with a double dash (--), and be sure not to have spaces around the equal sign. Example: `--debug-on-errors=true`

To get the list of all possible configuration options, type:

```
opp_run -h config
```

11.2 Cmdenv: the Command-Line Interface

The command line user interface is a small, portable and fast user interface that compiles and runs on all platforms. Cmdenv is designed primarily for batch execution.

Cmdenv simply executes some or all simulation runs that are described in the configuration file. If one run stops with an error message, subsequent ones will still be executed. The runs to be executed can be passed via command-line argument or in the ini file.

11.2.1 Example Run

When you run the Fifo example under Cmdenv, you should see something like this:

```
$ ./fifo -u Cmdenv -c Fifo1

OMNeT++ Discrete Event Simulation  (C) 1992-2008 Andras Varga, OpenSim Ltd.
Version: 4.0, edition: Academic Public License -- NOT FOR COMMERCIAL USE
See the license for distribution terms and warranty disclaimer
Setting up Cmdenv...
Loading NED files from .: 5

Preparing for running configuration Fifo1, run #0...
Scenario: $repetition=0
Assigned runID=Fifo1-0-20090104-12:23:25-5792
Setting up network 'FifoNet'...
Initializing...
Initializing module FifoNet, stage 0
Initializing module FifoNet.gen, stage 0
Initializing module FifoNet.fifo, stage 0
Initializing module FifoNet.sink, stage 0

Running simulation...
** Event #1   T=0   Elapsed: 0.000s (0m 00s)  0% completed
     Speed:     ev/sec=0   simsec/sec=0   ev/simsec=0
     Messages:  created: 2   present: 2   in FES: 1
** Event #232448   T=11719.051014922336   Elapsed: 2.003s (0m 02s)  3% completed
     Speed:     ev/sec=116050   simsec/sec=5850.75   ev/simsec=19.8351
     Messages:  created: 58114   present: 3   in FES: 2
...
** Event #7206882   T=360000.52066583684   Elapsed: 78.282s (1m 18s)  100% completed
     Speed:     ev/sec=118860   simsec/sec=5911.9   ev/simsec=20.1053
     Messages:  created: 1801723   present: 3   in FES: 2
```

```
<!> Simulation time limit reached -- simulation stopped.

Calling finish() at end of Run #0...
End.
```

As Cmdenv runs the simulation, it periodically prints the sequence number of the current event, the simulation time, the elapsed (real) time, and the performance of the simulation (how many events are processed per second; the first two values are 0 because there wasn't enough data for it to calculate yet). At the end of the simulation, the `finish()` methods of the simple modules are run, and the outputs from them are displayed. On my machine this run took 34 seconds. This Cmdenv output can be customized via `omnetpp.ini` entries. The output file `results/Fifo1-0.vec` contains vector data recorded during simulation (here, queueing times), and it can be processed using the IDE or other tools.

11.2.2 Command-Line Options

The command line environment allows you to specify more than one run by using the `-r 2,4,6..8` format. See 11.4 for more information about running simulation batches.

11.2.3 Cmdenv Ini File Options

`cmdenv-runs-to-execute` specifies which simulation runs should be executed. It accepts a comma-separated list of run numbers or run number ranges, e.g. `1,3..4,7..9`. If the value is missing, Cmdenv executes all runs that have ini file sections; if no runs are specified in the ini file, Cmdenv does one run. The `-r` command line option overrides this ini file setting.

Cmdenv can be executed in two modes, selected by the **`cmdenv-express-mode`** ini file option:

- **Normal** (non-express) mode is for debugging; detailed information will be written to the standard output (event banners, module output, etc).
- **Express** mode can be used for long simulation runs; only periodical status updates are displayed about the progress of the simulation.

`cmdenv-performance-display` affects express mode only: it controls whether to print performance information. Turning it on results in a 3-line entry printed on each update, containing ev/sec, simsec/sec, ev/simsec, number of messages created/still present/currently scheduled in FES.

For a full list of options, see the options beginning with `cmdenv-` in Appendix G.

11.2.4 Interpreting Cmdenv Output

When the simulation is running in "express" mode with detailed performance display enabled, Cmdenv periodically outputs a three-line status report about the progress of the simulation. The output looks like this:

```
...
** Event #250000   T=123.74354 ( 2m  3s)    Elapsed: 0m 12s
     Speed:     ev/sec=19731.6   simsec/sec=9.80713   ev/simsec=2011.97
     Messages:  created: 55532   present: 6553   in FES: 8
```

```
** Event #300000   T=148.55496 ( 2m 28s)    Elapsed: 0m 15s
     Speed:     ev/sec=19584.8   simsec/sec=9.64698   ev/simsec=2030.15
     Messages:  created: 66605   present: 7815   in FES: 7
...
```

The first line of the status display (beginning with `**`) contains:

- how many events have been processed so far
- the current simulation time (T), and
- the elapsed time (wall clock time) since the beginning of the simulation run.

The second line displays info about simulation performance:

- `ev/sec` indicates *performance*: how many events are processed in one real-time second. On one hand it depends on your hardware (faster CPUs process more events per second), and on the other hand it depends on the complexity (amount of calculations) associated with processing one event. For example, protocol simulations tend to require more processing per event than e.g. queueing networks, thus the latter produce higher ev/sec values. In any case, this value is independent of the size (number of modules) in your model.
- `simsec/sec` shows *relative speed* of the simulation, that is, how fast the simulation is progressing compared to real time, how many simulated seconds can be done in one real second. This value virtually depends on everything: on the hardware, on the size of the simulation model, on the complexity of events, and the average simulation time between events as well.
- `ev/simsec` is the *event density*: how many events are there per simulated second. Event density only depends on the simulation model, regardless of the hardware used to simulate it: in a cell-level ATM simulation you will have very hight values (10^9), while in a bank teller simulation this value is probably well under 1. It also depends on the size of your model: if you double the number of modules in your model, you can expect the event density double, too.

The third line displays the number of messages, and it is important because it may indicate the 'health' of your simulation.

- `Created`: total number of message objects created since the beginning of the simulation run. This does not mean that this many message object actually exist, because some (many) of them may have been deleted since then. It also does not mean that *you* created all those messages – the simulation kernel also creates messages for its own use (e.g. to implement `wait()` in an `activity()` simple module).
- `Present`: the number of message objects currently present in the simulation model, that is, the number of messages created (see above) minus the number of messages already deleted. This number includes the messages in the FES.
- `In FES`: the number of messages currently scheduled in the Future Event Set.

The second value, the number of messages present, is more useful than perhaps one would initially think. It can be an indicator of the 'health' of the simulation; if it is growing steadily,

then either you have a memory leak and losing messages (which indicates a programming error), or the network you simulate is overloaded and queues are steadily filling up (which might indicate wrong input parameters).

Of course, if the number of messages does not increase, it does not mean that you do *not* have a memory leak (other memory leaks are also possible). Nevertheless the value is still useful, because by far the most common way of leaking memory in a simulation is by not deleting messages.

11.3 Tkenv: the Graphical User Interface

Tkenv is a portable graphical windowing user interface. Tkenv supports interactive execution of the simulation, tracing and debugging. Tkenv is recommended in the development stage of a simulation and for presentation purposes, since it allows one to get a detailed picture of the state of simulation at any point of execution and to follow what happens inside the network.

> **NOTE:** This section only covers the command-line and configuration options of Tkenv; the user interface is described in the Tkenv chapter of the OMNeT++ User Guide.

11.3.1 Command-Line and Configuration Options

A simulation program built with Tkenv accepts all the general command line options. Additionally, the `-c configname` and `-r runnumber` options can be used to preselect a single run for execution; that is, these options suppress the initial run selection dialog.

Tkenv configuration options:

- **`tkenv-default-config`**: Specifies which config Tkenv should set up automatically on startup. The default is to ask the user. This option is equivalent to the `-c` command-line option.
- **`tkenv-default-run`**: Specifies which run (of the default config, see tkenv-default-config) Tkenv should set up automatically on startup. The default is to ask the user. This option is equivalent to the `-r` command-line option.
- **`tkenv-extra-stack`**: Specifies the extra amount of stack that is reserved for each activity() simple module when the simulation is run under Tkenv.
- **`tkenv-image-path`**: Specifies the path for loading module icons.
- **`tkenv-plugin-path`**: Specifies the search path for Tkenv plugins. Tkenv plugins are .tcl files that get evaluated on startup.

Tkenv-specific configuration options can also be specified on the command line by prefixing them with two dashes (e.g `--tkenv-option=value`). See Appendix G for the list of possible configuration options.

11.4 Batch Execution

Once your model works reliably, you will 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.

Before running simulation batches, you must set a condition to stop your simulation. This is usually a time limit set by the **sim-time-limit** configuration option, but you can limit your simulation by using wall clock time (**cpu-time-limit**) or by directly ending a simulation with an API call if some condition is true.

11.4.1 Using Cmdenv

To execute more than one run using Cmdenv, use the `-r` option and specify the runs in a comma separated format `1,2,4,9..11`, or you may leave out the `-r` option to execute all runs in the experiment.

> **WARNING:** Although it is very convenient, we do not recommend that you use this method for running simulation batches. Specifying more than one run number would run those simulations in the same process. This method is more prone to C++ programming errors. A failure in a single run may abort execution (segfault) or invalidate the results of subsequent runs. If you want to execute more than one run, we recommend that you run each of them in a separate process; you can use the `opp_runall` program for this purpose.

11.4.2 Using Shell Scripts

The following script executes a simulation named `wireless` several times, with parameters for the different runs given in the `runs.ini` file.

Before you execute your simulation batch, you may check how many runs are available in the configuration you are using. Use the `-x config` command line option to print the number of runs or add the `-g` to get more details.

```
#! /bin/sh
./wireless -f runs.ini -r 1
./wireless -f runs.ini -r 2
./wireless -f runs.ini -r 3
./wireless -f runs.ini -r 4
...
./wireless -f runs.ini -r 10
```

To run the above script, type it in a text file called e.g. `run`, give it `x` (executable) permission using `chmod`, then you can execute it by typing `./run`:

```
$ chmod +x run
$ ./run
```

You can simplify the above script by using a *for* loop. In the example below, the variable `i` iterates through the values of list given after the `in` keyword. It is very practical, since you can leave out or add runs, or change the order of runs by simply editing the list – to demonstrate this, we skip run 6, and include run 15 instead.

```
#! /bin/sh
for i in 3 2 1 4 5 7 15 8 9 10; do
   ./wireless -f runs.ini -r $i
done
```

If you have many runs, you can use a C-style loop:

```
#! /bin/sh
for ((i=1; $i<50; i++)); do
   ./wireless -f runs.ini -r $i
done
```

11.4.3 Using opp_runall

OMNeT++ has a utility program called `opp_runall` which allows you to execute a simulation batch in command line mode. You must specify the whole command line you would use to run your batch in Cmdenv. There are advantages to running your batches this way:

- Each simulation run executes in a separate operating system process. This means that a crash because of a programming error does not affect the outcome of the other runs. They are totally independent of each other.
- If you happen to have a multi core/processor machine, you can take advantage of the processing power by running sevaral runs in parallel.

The command basically creates a makefile which contains a separate target for each run. By default the makefile will be executed causing each target to run. You can give additional options to the `opp_runall` command to activate parallel building. The `-j` option can be used to specify the maximum number of parallel runs allowed.

WARNING: Use the parallel execution option only if you have enough memory to run several simulations side by side. If you run out of memory your operating system will start swapping, and the overall performance of the system will be greatly reduced. Always specify the number of processes after the `-j` option, otherwise the `make` program will try to start *all* runs at the same time. As a rule of thumb: if you have 4 cores (and enough memory), use `-j4`.

The form of the command is:

```
opp_runall -j2 ./aloha -u Cmdenv -c PureAlohaExperiment -r 0..23
```

You can use the `-x ConfigName -g` command line options with your simulation to check the number of available runs.

Using the `--export filename` option only generates the `makefile`, but does not start it. You can run your batch later by invoking the generated makefile.

11.5 Akaroa Support: Multiple Replications in Parallel

11.5.1 Introduction

Typical simulations are Monte-Carlo simulations: they use (pseudo-)random numbers to drive the simulation model. For the simulation to produce statistically reliable results, one has to

carefully consider the following:

- When the initial transient is over, when can we start collecting data? We usually don't want to include the initial transient when the simulation is still "warming up."
- When can we stop the simulation? We want to wait long enough so that the statistics we are collecting can "stabilize", or reach the required sample size to be statistically trustable.

Neither question is trivial to answer. One might just suggest to wait "very long" or "long enough". However, this is neither simple (how do you know what is "long enough"?) nor practical (even with today's high speed processors simulations of modest complexity can take hours, and one may not afford multiplying runtimes by, say, 10, "just to be safe.") If you need further convincing, please read [PJL02] and be horrified.

A possible solution is to look at the statistics while the simulation is running, and decide at runtime when enough data have been collected for the results to have reached the required accuracy. One possible criterion is given by the confidence level, more precisely, by its width relative to the mean. But ex ante it is unknown how many observations have to be collected to achieve this level – it must be determined at runtime.

11.5.2 What Is Akaroa

Akaroa [EPM99] addresses the above problem. According to its authors, Akaroa (Akaroa2) is a "fully automated simulation tool designed for running distributed stochastic simulations in MRIP scenario" in a cluster computing environment.

MRIP stands for *Multiple Replications in Parallel*. In MRIP, the computers of the cluster run independent replications of the whole simulation process (i.e. with the same parameters but different seed for the RNGs (random number generators)), generating statistically equivalent streams of simulation output data. These data streams are fed to a global data analyser responsible for analysis of the final results and for stopping the simulation when the results reach a satisfactory accuracy.

The independent simulation processes run independently of one another and continuously send their observations to the central analyser and control process. This process *combines* the independent data streams, and calculates from these observations an overall estimate of the mean value of each parameter. Akaroa2 decides by a given confidence level and precision whether it has enough observations or not. When it judges that is has enough observations it halts the simulation.

If n processors are used, the needed simulation execution time is usually n times smaller compared to a one-processor simulation (the required number of observations are produced sooner). Thus, the simulation would be sped up approximately in proportion to the number of processors used and sometimes even more.

Akaroa was designed at the University of Canterbury in Christchurch, New Zealand and can be used free of charge for teaching and non-profit research activities.

11.5.3 Using Akaroa with OMNeT++

Akaroa

Before the simulation can be run in parallel under Akaroa, you have to start up the system:

- Start `akmaster` running in the background on some host.
- On each host where you want to run a simulation engine, start `akslave` in the background.

Each `akslave` establishes a connection with the `akmaster`.

Then you use `akrun` to start a simulation. `akrun` waits for the simulation to complete, and writes a report of the results to the standard output. The basic usage of the `akrun` command is:

```
akrun -n num_hosts command [argument..]
```

where *command* is the name of the simulation you want to start. Parameters for Akaroa are read from the file named `Akaroa` in the working directory. Collected data from the processes are sent to the `akmaster` process, and when the required precision has been reached, `akmaster` tells the simulation processes to terminate. The results are written to the standard output.

The above description is not detailed enough help you set up and successfully use Akaroa – for that you need to read the Akaroa manual.

Configuring OMNeT++ for Akaroa

First of all, you have to compile OMNeT++ with Akaroa support enabled.

The OMNeT++ simulation must be configured in `omnetpp.ini` so that it passes the observations to Akaroa. The simulation model itself does not need to be changed – it continues to write the observations into output vectors (`cOutVector` objects, see chapter 7). You can place some of the output vectors under Akaroa control.

You need to add the following to `omnetpp.ini`:

```
[General]
rng-class = "cAkaroaRNG"
outputvectormanager-class = "cAkOutputVectorManager"
```

These lines cause the simulation to obtain random numbers from Akaroa, and allows data written to selected output vectors to be passed to Akaroa's global data analyser. [1]

Akaroa's RNG is a Combined Multiple Recursive pseudorandom number generator (CMRG) with a period of approximately 2^{191} random numbers, and provides a unique stream of random numbers for every simulation engine.

NOTE: It is vital that you obtain random numbers from Akaroa; otherwise, all simulation processes will run with the same RNG seeds, and produce exactly the same results.

Then you need to specify which output vectors you want to be under Akaroa control (by default, none of them are). You can use the `*`, `**` wildcards (see section 10.3.1) to place certain vectors under Akaroa control.

```
<modulename>.<vectorname1>.with-akaroa = true
<modulename>.<vectorname2>.with-akaroa = true
```

[1] For more details on the plugin mechanism these settings make use of, see section 16.1.

Using Shared File Systems

It is usually practical to have the same physical disk mounted (e.g. via NFS or Samba) on all computers in the cluster. However, because all OMNeT++ simulation processes run with the same settings, they would overwrite each other's output files (e.g. `omnetpp.vec`, `omnetpp.sca`). Your can prevent this from happening using the **`fname-append-host`** ini file entry:

```
[General]
fname-append-host = true
```

When turned on, it appends the host name to the names of the output files (output vector, output scalar, snapshot files).

11.6 Troubleshooting

11.6.1 Unrecognized Configuration Option

If you receive an error message about unrecognized configuration options you may use `-h config` or `-h configdetails` options to display all possible configuration options and their descriptions.

11.6.2 Stack Problems

"Stack violation (*FooModule* stack too small?) in module *bar.foo*"

OMNeT++ detected that the module has used more stack space than it has allocated. The solution is to increase the stack for that module type. You can call the `getStackUsage()` from `finish()` to find out actually how much stack the module used.

"Error: Cannot allocate *nn* bytes stack for module *foo.bar*"

The resolution depends on whether you are using OMNeT++ on Unix or on Windows.

Unix. If you get the above message, you have to increase the total stack size (the sum of all coroutine stacks). You can do so in `omnetpp.ini`:

```
[General]
total-stack = 2MiB
```

There is no performance penalty if you set **`total-stack`** 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 the coroutine stack size, and with 1MiB being the default, it is easy to run out of the 2GiB possible address space (2GiB/1MiB=2048).

A more detailed explanation follows. Each fiber has its own stack, by default 1MiB (this is the "reserved" stack space – i.e. reserved in the address space, but not the full 1MiB is actually "committed", i.e. has physical memory assigned to it). This means that a 2GiB 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 1MiB reserved stack size (RSS) must be set to a smaller value: the coroutine stack size requested for the module, plus the `extra-stack-kb` amount for Cmdenv/Tkenv – which makes about 16K with Cmdenv, and about 48K when using Tkenv. Unfortunately, the CreateFiber() Win32 API doesn't allow the RSS to be specified. The more advanced CreateFiberEx() API which accepts RSS as parameter is unfortunately only available from Windows XP.

The alternative is the stacksize parameter stored in the EXE header, which can be set via the STACKSIZE .def file parameter, via the /stack linker option, or on an existing executable using the editbin /stack utility. This parameter specifies a common RSS for the main program stack, fiber and thread stacks. 64K should be enough. This is the way the 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, 8MiB). 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 are 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 don't 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, if you are 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.

11.6.3 Memory Leaks and Crashes

The most common problems in C++ are associated with memory allocation (usage of `new` and `delete`):

- *memory leaks,* that is, forgetting to delete objects or memory blocks no longer used;
- *crashes,* usually due to referring to an already deleted object or memory block, or trying to delete one for a second time;
- *heap corruption* (eventually leading to crash) due to overrunning allocated blocks, i.e. writing past the end of an allocated array.

The most common cause of memory leaks in OMNeT++ simulations is forgetting to delete messages. Both Tkenv and Cmdenv are able to display the number of messages currently in the simulation, helping you to determine if you have such a memory leak; see section 11.2.4. If you find that the number of messages is steadily increasing, you need to find where the message objects are located. 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 number of messages is stable, it is still possible you are leaking other `cOwnedObject`-based objects. You can find them using Tkenv's *Inspect|From list of all objects...* function as well.

If you are leaking non-`cOwnedObject`-based objects or just memory blocks (structs, arrays, etc., allocated by `new`), you will not be able to find them via Tkenv. You will probably need a specialized memory debugging tool like the ones described below.

Memory Debugging Tools

If you suspect that you may have memory allocation problems (crashes associated with double-deletion or accessing already deleted block, or memory leaks), you can use specialized tools to track them down.

By far the most efficient, most robust and most versatile tool is *Valgrind*, originally developed for debugging KDE.

Other memory debuggers are *NJAMD*, *MemProf*, *MPatrol*, *dmalloc* and *ElectricFence*. Most of the above tools support tracking down memory leaks as well as detecting double deletion, writing past the end of an allocated block, etc.

A proven commercial tool is *Rational Purify*. It has a good reputation and has proved its usefulness many times.

11.6.4 Simulation Executes Slowly

Check the following if you think your simulation is running too slowly:

- Turn on express mode with the `cmdenv-express-mode=true` configuration option.
- Be sure that event logging is turned off (`record-eventlog=false` configuration option).
- Turn of vector file recording if you don't absolutely need it (`**.vector-recording=false`).
- If you are running under Tkenv, disable animation features, close inspectors, hide the timeline, hide object tree, turn off log filtering.
- Compile your code as release instead of debug (in some cases this can give you 5x speedup)

What can you do if the simulation executes much slower than you expect? The best advice that can be given here is that you should **use a good profiler** to find out how much time is spent in each part of the program. Do not make the mistake of omitting this step, thinking that you know which part is slow! Even for experienced programmers, a profiling session is all too often full of surprises. It often turns out that lots of CPU time is spent in completely innocent-looking statements, while big and complex algorithms don't take nearly as much time as you expected. *Don't assume anything – profile before you optimize!*

A great profiler on Linux is the *Valgrind*-based *callgrind*, and its visualizer *KCachegrind*. Unfortunately it won't be ported to Windows anytime soon. On Windows, you are out of luck – commercial products may help, or, port your simulation to Linux. The latter goes usually much more smoothly than one would expect.

Chapter 12

Result Recording and Analysis

12.1 Result Recording

OMNeT++ provides built-in support for recording simulation results, via *output vectors* and *output scalars*. Output vectors are time series data, recorded from simple modules or channels. You can use output vectors to record end-to-end delays or round trip times of packets, queue lengths, queueing times, module state, link utilization, packet drops, etc. – anything that is useful to get a full picture of what happened in the model during the simulation run.

Output scalars are summary results, computed during the simulation and written out when the simulation completes. A scalar result may be an (integer or real) number, or may be a statistical summary comprised of several fields such as count, mean, standard deviation, sum, minimum, maximum, etc., and optionally histogram data.

Results may be collected and recorded in two ways:

1. Based on the signal mechanism, using declared statistics;
2. Directly from C++ code, using the simulation library

The second method has been the traditional way of recording results. The first method, based on signals and declared statistics, was introduced in OMNeT++ 4.1, and it is preferable because it allows you to always record the results in the form you need, without requiring heavy instrumentation or continuous tweaking of the simulation model.

12.1.1 Using Signals and Declared Statistics

This approach combines the signal mechanism (see 4.14) and NED properties (see 3.12) in order to de-couple the generation of results from their recording, thereby providing more flexibility in what to record and in which form. The details of the solution have been described in section 4.15 in detail; here we just give a short overview.

Statistics are declared in the NED files with the `@statistic` property, and modules emit values using the signal mechanism. The simulation framework records data by adding special result file writer listeners to the signals. By being able to choose what listeners to add, the user can control what to record in the result files and what computations to apply before recording. The aforementioned section 4.15 also explains how to instrument simple modules and channels for signals-based result recording.

The signals approach allows for calculation of aggregate statistics (such as the total number of packet drops in the network) and for implementing a warm-up period without support from module code. It also allows you to write dedicated statistics collection modules for the simulation, also without touching existing modules.

The same configuration options that were used to control result recording with `cOutVector` and `recordScalar()` also work when utilizing the signals approach, and there are extra configuration options to make the additional possibilities accessible.

12.1.2 Direct Result Recording

With this approach, scalar and statistics results are collected in class variables inside modules, then recorded in the finalization phase via `recordScalar()` calls. Vectors are recorded using `cOutVector` objects. To record more details, like the minimum/maximum value or the standard deviation, `cStdDev` and `cWeightedStdDev` can be used, and for recording the distribution there are histogram and other distribution estimation classes (`cDoubleHistogram`, `cLongHistogram`, `cPSquare`, `cKSplit`, and others). These classes are described in sections 7.8 and 7.9. Recording of individual vectors, scalars and statistics can be enabled or disabled via the configuration (ini file), and it is also the place to set up recording intervals for vectors.

The drawback of recording results directly from modules is that result recording is hardcoded in modules, and even simple requirement changes (e.g. record the average delay instead of each delay value, or vice versa) requires either code change or an excessive amount of result collection code in the modules.

12.2 Configuring Result Collection

12.2.1 Configuring Signal-Based Statistics Recording

Signal-based statistics recording has been designed so that it can be easily configured to record a "default minimal" set of results, a "detailed" set of results, and a custom set of results (by modifying the previous ones, or defined from scratch).

Recording can be tuned with the **`result-recording-modes`** per-object configuration option. The "object" here is the statistic, which is identified by the full path (hierarchical name) of the module or connection channel object in question, plus the name of the statistic (which is the "index" of `@statistic` property, i.e. the name in the square brackets). Thus, configuration keys have the syntax *<module-full-path>.<statistic-name>*`.result-recording-modes=`.

The **`result-recording-modes`** option accepts one or more items as value, separated by comma. An item may be a result recording mode (surprise!), and two words with a special meaning, `default` and `all`:

- A *result recording mode* means any item that may occur in the `record` key of the `@statistic` property; for example, `count`, `sum`, `mean`, `vector((count-1)/2)`.
- **`default`** stands for the set of non-optional items from the `@statistic` property's `record` list, that is, those without question marks.
- **`all`** means all items from the `@statistic` property's `record` list, including the ones with question marks.

The default value is `default`.

A lone "-" as option value disables all recording modes.

Recording mode items in the list may be prefixed with "+" or "-" to add/remove them from the set of result recording modes. The initial set of result recording modes is `default`; if the first item is prefixed with "+" or "-", then that and all subsequent items are understood as modifying the set; if the first item does not start with with "+" or "-", then it replaces the set, and further items are understood as modifying the set.

This sounds more complicated than it is; an example will make it clear. Suppose we are configuring the following statistic:

```
@statistic[foo](record=count,mean,max?,vector?);
```

With the following the ini file lines (see results in comments):

```
**.result-recording-modes = default   # --> count, mean
**.result-recording-modes = all       # --> count, mean, max
**.result-recording-modes = -         # --> none
**.result-recording-modes = mean      # --> only mean (disables 'default')
**.result-recording-modes = default,-vector,+histogram # --> count,mean,histogram
**.result-recording-modes = -vector,+histogram       # --> same as above
**.result-recording-modes = all,-vector,+histogram   # --> count,mean,max,histogram
```

Here is another example which shows how to write a more specific option key. The following line applies to `queueLength` statistics of `fifo[]` submodule vectors anywhere in the network:

```
**.fifo[*].queueLength.result-recording-modes = +vector   # default modes plus vector
```

In the result file, the recorded scalars will be suffixed with the recording mode, i.e. the mean of `queueingTime` will be recorded as `queueingTime:mean`.

> **NOTE:** Signal-based statistics recording forms a layer above the normal scalar and vector recording infrastructure, so options like **`scalar-recording`**, **`vector-recording`**, **`vector-recording-intervals`** also affect it. These options are described in the following sections.

12.2.2 Warm-up Period

The **`warmup-period`** option specifies the length of the initial warm-up period. When set, results belonging to the first x seconds of the simulation will not be recorded into output vectors, and will not be counted into the calculation of output scalars. This option is useful for steady-state simulations. The default is 0s (no warmup period).

Example:

```
warmup-period = 20s
```

Results recorded via signal-based statistics automatically obey the warm-up period setting, but modules that compute and record scalar results manually (via `recordScalar()`) need to be modified so that they take the warm-up period into account.

> **NOTE:** When configuring a warm-up period, make sure that modules that compute and record scalar results manually via `recordScalar()` actually obey the warm-up period in the C++ code.

The warm-up period is available via the `getWarmupPeriod()` method of the `simulation` object, so the C++ code that updates the corresponding state variables needs to be surrounded with an *if* statement:

Old:

```
dropCount++;
```

New:

```
if (simTime() >= simulation.getWarmupPeriod())
    dropCount++;
```

12.2.3 Result File Names

Simulation results are recorded into *output scalar files* that actually hold statistics results as well, and *output vector files*. The usual file extension for scalar files is `.sca`, and for vector files `.vec`.

Every simulation run generates a single scalar file and a vector file. The file names can be controlled with the **`output-vector-file`** and **`output-scalar-file`** options. These options rarely need to be used, because the default values are usually fine. The defaults are:

```
output-vector-file = "${resultdir}/${configname}-${runnumber}.vec"
output-scalar-file = "${resultdir}/${configname}-${runnumber}.sca"
```

Here, `${resultdir}` is the value of the **`result-dir`** configuration option which defaults to `results/`, and `${configname}` and `${runnumber}` are the name of the configuration name in the ini file (e.g. `[Config PureAloha]`), and the run number. Thus, the above defaults generate file names like `results/PureAloha-0.vec`, `results/PureAloha-1.vec`, and so on.

> **NOTE:** In OMNeT++ 3.x, the default result file names were `omnetpp.vec` and `omnetpp.sca`, and scalar files were always appended to, rather than being overwritten as in the 4.x version. When needed, the old behavior for scalar files can be turned back on by setting `output-scalar-file-append=true` in the configuration.

12.2.4 Configuring Scalar Results

Recording results into the scalar file can be turned off globally by adding the following line to the ini file. Any existing file with the same name will still be removed before the simulation starts.

```
**.scalar-recording = false
```

Recording scalar results can be enabled or disabled individually, using patterns. The syntax of the configuration option is *<module-full-path>*`.`*<scalar-name>*`.scalar-recording=`*true/false*, where both *<module-full-path>* and *<scalar-name>* may contain wildcards (see 10.3.1). *<scalar-name>* is the signal name, or the string passed to the `recordScalar()` call. By default, the recording of all scalars is enabled.

The following example turns off recording all scalar results, except end-to-end delays and those produced by TCP modules:

```
**.tcp.**.scalar-recording = true
**.endToEndDelay.scalar-recording = true
**.scalar-recording = false
```

12.2.5 Configuring Output Vectors

Recording output vector results can be turned off globally by adding the following line to the ini file. Any existing file with the same name will still be removed before the simulation starts.

```
**.vector-recording = false
```

The size of output vector files can easily reach the magnitude of several hundred megabytes, but very often, only some of the recorded statistics are interesting to the analyst. OMNeT++ allows you to control which vectors you want to record, and to specify one or more collection intervals.

Output vectors can be configured with the **vector-recording** and **vector-recording-intervals** per-object options. The syntax of the configuration options are *<module-full-path>.<vector-name>.*`vector-recording=`*true/false*, and *<module-full-path>.<vector-name>.*`vector-recording-intervals=`*<intervals>*, where both *<module-full-path>* and *<vector-name>* may contain wildcards (see 10.3.1). *<vector-name>* is the signal name, or the name string of the `cOutVector` object. By default, all output vectors are turned on for the whole duration the simulation.

The following example only records the `queueLength` vectors and `endToEndDelay` in `voiceApp` modules, and turns off the rest:

```
**.queueLength.vector-recording = true
**.voiceApp.endToEndDelay.vector-recording = true
**.vector-recording = false
```

For the **vector-recording-intervals** option, one can specify one or more intervals in the *<startTime>..<stopTime>* syntax, separated by comma. *<startTime>* or *<stopTime>* need to be given with measurement units, and both can be omitted to denote the beginning and the end of the simulation, respectively.

The following example limits all vectors to three intervals, except `dropCount` vectors which will be recorded during the whole simulation run:

```
**.dropCount.vector-recording-intervals = 0..
**.vector-recording-intervals = 0..1000s, 5000s..6000s, 9000s..
```

A third per-vector configuration option is **vector-record-eventnumbers**, which specifies whether to record event numbers for an output vector. (Simulation time and value are always recorded. Event numbers are needed by the Sequence Chart Tool, for example.) Event number recording is enabled by default; it may be turned off to save disk space.

```
**.vector-record-eventnumbers = false
```

If the (default) `cIndexedFileOutputVectorManager` class is used to record output vectors, there are two more options to fine-tune its resource usage. `output-vectors-memory-limit` specifies the total memory that can be used for buffering output vectors. Larger values produce less fragmented vector files (i.e. cause vector data to be grouped into larger chunks), and therefore allow more efficient processing later. `vector-max-buffered-values` specifies the maximum number of values to buffer per vector, before writing out a block into the output vector file. The default is no per-vector limit (i.e. only the total memory limit is in effect.)

12.2.6 Saving Parameters as Scalars

When you are running several simulations with different parameter settings, you'll usually want to refer to selected input parameters in the result analysis as well – for example when

drawing a throughput (or response time) versus load (or network background traffic) plot. Average throughput or response time numbers are saved into the output scalar files, and it is useful for the input parameters to get saved into the same file as well.

For convenience, OMNeT++ automatically saves the iteration variables into the output scalar file if they have numeric value, so they can be referred to during result analysis.

WARNING: If an iteration variable has non-numeric value, it will not be recorded automatically and cannot be used during analysis. This can happen unintentionally if you specify units inside an iteration variable list:

```
**.param = exponential( ${mean=0.2s, 0.4s, 0.6s} )   #WRONG!
**.param = exponential( ${mean=0.2, 0.4, 0.6}s )     #OK
```

Module parameters can also be saved, but this has to be requested by the user, by configuring `param-record-as-scalar=true` for the parameters in question. The configuration key is a pattern that identifies the parameter, plus `.param-record-as-scalar`. An example:

```
**.host[*].networkLoad.param-record-as-scalar = true
```

This looks simple enough, however there are three pitfalls: non-numeric parameters, too many matching parameters, and random-valued volatile parameters.

First, the scalar file only holds numeric results, so non-numeric parameters cannot be recorded – that will result in a runtime error.

Second, if wildcards in the pattern match too many parameters, that might unnecessarily increase the size of the scalar file. For example, if the `host[]` module vector size is 1000 in the example below, then the same value (3) will be saved 1000 times into the scalar file, once for each host.

```
**.host[*].startTime = 3
**.host[*].startTime.param-record-as-scalar = true  # saves "3" once for each host
```

Third, recording a random-valued volatile parameter will just save a random number from that distribution. This is rarely what you need, and the simulation kernel will also issue a warning if this happens.

```
**.interarrivalTime = exponential(1s)
**.interarrivalTime.param-record-as-scalar = true  # wrong: saves random values!
```

These pitfalls are quite common in practice, so it is usually better to rely on the iteration variables in the result analysis. That is, one can rewrite the above example as

```
**.interarrivalTime = exponential( ${mean=1}s )
```

and refer to the `$mean` iteration variable instead of the interarrivalTime module parameter(s) during result analysis. `param-record-as-scalar=true` is not needed, because iteration variables are automatically saved into the result files.

12.2.7 Recording Precision

Output scalar and output vector files are text files, and floating point values (`double`s) are recorded into it using `fprintf()`'s `"%g"` format. The number of significant digits can be configured using the **`output-scalar-precision`** and **`output-vector-precision`** configuration options.

The default precision is 12 digits. The following has to be considered when setting a different value:

IEEE-754 doubles are 64-bit numbers. The mantissa is 52 bits, which is roughly equivalent to 16 decimal places (52*log(2)/log(10)). However, due to rounding errors, usually only 12..14 digits are correct, and the rest is pretty much random garbage which should be ignored. However, when you convert the decimal representation back into a `double` for result processing, an additional small error will occur, because 0.1, 0.01, etc. cannot be accurately represented in binary. This conversion error is usually smaller than what that the `double` variable already had before recording into the file. However, if it is important, you can eliminate this error by setting the recording precision to 16 digits or more (but again, be aware that the last digits are garbage). The practical upper limit is 17 digits, setting it higher doesn't make any difference in `fprintf()`'s output.

Errors resulting from converting to/from decimal representation can be eliminated by choosing an output vector/output scalar manager class which stores `double`s in their native binary form. The appropriate configuration options are **`outputvectormanager-class`** and **`outputvectormanager-class`**. For example, `cMySQLOutputScalarManager` and `cMySQLOutputScalarManager` provided in `samples/database` fulfill this requirement.

However, before worrying too much about rounding and conversion errors, consider the *real* accuracy of your results:

- in real life, it is very difficult to measure quantities (weight, distance, even time) with more than a few digits of precision. What precision are your input data? For example, if you approximate inter-arrival time as *exponential(0.153)* when the mean is really *0.152601...* and the distribution is not even exactly exponential, you are already starting out with a bigger error than rounding can cause.
- the simulation model is itself an approximation of real life. How much error do the (known and unknown) simplifications cause in the results?

12.3 Overview of the Result File Formats

Both output vector and scalar files are textual, line-oriented files. The advantage of a text-based format is that it is very accessible with a wide range of tools and languages. The format of result files is documented in detail in Appendix H.

By default, each file contains data from one run only.

Result files start with a header that contains several attributes of the simulation run: a reasonably globally unique run ID, the network NED type name, the experiment-measurement-replication labels, the values of iteration variables and the repetition counter, the date and time, the host name, the process id of the simulation, random number seeds, configuration options, and so on. These data can be useful during result processing, and increase the reproducibility of the results.

Vectors are recorded into a separate file for practical reasons: vector data usually consume several magnitudes more disk space than scalars.

12.3.1 Output Vector Files

All output vectors from a simulation run are recorded into the same file. The following sections describe the format of the file, and how to process it.

An example file fragment (without header):

```
...
vector 1   net.host[12]  responseTime  TV
1  12.895  2355.66
1  14.126  4577.66664666
vector 2   net.router[9].ppp[0] queueLength  TV
2  16.960  2
1  23.086  2355.66666666
2  24.026  8
...
```

There two types of lines: vector declaration lines (beginning with the word `vector`), and data lines. A *vector declaration line* introduces a new output vector, and its columns are: vector Id, module of creation, name of `cOutVector` object, and multiplicity (usually 1). Actual data recorded in this vector are on *data lines* which begin with the vector Id. Further columns on data lines are the simulation time and the recorded value.

Since OMNeT++ 4.0, vector data are recorded into the file clustered by output vectors, which, combined with index files, allows much more efficient processing. Using the index file, tools can extract particular vectors by reading only those parts of the file where the desired data are located, and do not need to scan through the whole file linearly.

12.3.2 Scalar Result Files

Fragment of an output scalar file (without header):

```
...
scalar "lan.hostA.mac" "frames sent"  99
scalar "lan.hostA.mac" "frames rcvd"  3088
scalar "lan.hostA.mac" "bytes sent"   64869
scalar "lan.hostA.mac" "bytes rcvd"   3529448
...
```

Every scalar generates one `scalar` line in the file.

Statistics objects (`cStatictic` subclasses such as `cStdDev`) generate several lines: mean, standard deviation, etc.

12.4 The Analysis Tool in the Simulation IDE

The Simulation IDE provides an Analysis Tool for analysis and visualization of simulation results. The Analysis Tool lets you load several result files at once, and presents their contents somewhat like a database. You can browse the results, select the particular data you are interested in (scalars, vectors, histograms), apply processing steps, and create various charts or plots from them. Data selection, processing and charting steps can be freely combined, resulting in a high degree of freedom. These steps are grouped into and stored as "recipes", which get automatically re-applied when new result files are added or existing files are replaced. This automation spares the user lots of repetitive manual work, without resorting to scripting.

The Analysis Tool is covered in detail in the User Guide.

12.5 Scave Tool

Much of the IDE Analysis Tool's functionality is available on the command line as well, via the `scavetool` program. `scavetool` is suitable for filtering and basic processing of result files, and exporting the result in various formats digestible for other tools. `scavetool` has no graphics capabilities, but it can be used to produce files that can be directly plotted with other tools like gnuplot (see 12.6.4).

When `scavetool` is invoked without arguments, it prints usage information:

```
scavetool <command> [options] <file>...
```

12.5.1 The *filter* Command

The *filter* command allows you to filter and/or convert result files.

A filter can be specified with the *-p <filter>* option. The filter is one or more *<pattern>* or *<fieldname>(<pattern>)* expressions connected with `AND`, `OR` and `NOT` operators; a naked *<pattern>* is understood as `name(`*<pattern>*`)`. For example, the filter `"module(**.sink) AND name(delay)"` (or just `"module(**.sink) AND delay"`) selects the `delay` vectors from all `sink` modules.

The possible field names are:

- **file**: full path of the result file
- **run**: run identifier
- **module**: module name
- **name**: vector name
- **attr:<runAttribute>**: value of an attribute of the run, e.g. `experiment`, `datetime` or `network`
- **param:<moduleParameter>**: value of the parameter in the run

Processing operations can be applied to vectors by the *-a <function>(<parameterlist>)* option. You can list the available functions and their parameters with the *info* command.

The name and format of the output file can be specified with the *-O <file>* and *-F <formatname>* options, where the format name is one of the following:

- **vec**: vector file (default)
- **csv**: CSV file
- **octave**: Octave text file
- **matlab**: Matlab script file

The following example writes the window-averaged queuing times stored in `in.vec` into `out.vec`:

```
scavetool filter -p "queuing time" -a winavg(10) -O out.vec in.vec
```

The next example writes the queueing and transmission times of `sink` modules into CSV files. It generates a separate file for each vector, named `out-1.csv`, `out-2.csv`, etc.

```
scavetool filter -p "module(**.sink) AND
                     (\"queueing time\" OR \"transmission time\")"
                 -O out.csv -F csv in.vec
```

The generated CSV files contain a header and two columns:

```
time,"Queue.sink.queueing time"
2.231807576851,0
7.843802235089,0
15.797137536721,3.59449
21.730758362277,6.30398
[...]
```

12.5.2 The *index* Command

If the index file was deleted or the vector file was modified, you need to rebuild the index file before running the filter command:

```
scavetool index Aloha-1.vec
```

Normally the vector data is written in blocks into the vector file. However, if the vector file was generated by an older version of the `cOutputVectorManager`, it might not be so. In this case you have to specify the -r option to rearrange the records of the vector file, otherwise the index file would be too big and the indexing inefficient.

12.5.3 The *summary* Command

The *summary* command reports the list of statistics names, module names, run ids, configuration names in the given files to the standard output.

```
scavetool summary Aloha-1.vec
```

12.6 Alternative Statistical Analysis and Plotting Tools

There are several programs and packages in addition to the OMNeT++ IDE and `scavetool` that can also be used to analyze simulation results, and create various plots and charts from them.

> **HINT:** Our recommendation is GNU R because of its features, its popularity, and the existence of an extension package written specifically for OMNeT++ result processing.

12.6.1 GNU R

R is a free software environment for statistical computing and graphics. R has an excellent programming language and powerful plotting capabilities, and it is supported on all major operating systems and platforms.

R is widely used for statistical software development and data analysis. The program uses a command line interface, though several graphical user interfaces are available.

HINT: An R package for OMNeT++ result processing is available from `https://github.com/omnetpp/omnetpp-resultfiles/wiki`. The package supports loading the contents of OMNeT++ result files into R, organizing the data and creating various plots and charts. The package is well documented, and the web site offers a Tutorial, a Tips page, a tutorial for the Scalar Lattice GUI package, and other information.

Several other OMNeT++-related packages such as SimProcTC and Syntony already use R for data analysis and plotting.

12.6.2 NumPy, SciPy and MatPlotLib

NumPy and SciPy are numerical and scientific computing packages for the Python programming language, and MatPlotlib is a plotting library (also for Python).

MatPlotlib provides a "pylab" API designed to closely resemble that of MATLAB, thereby making it easy to learn for experienced MATLAB users. Matplotlib is distributed under a BSD-style license.

12.6.3 MATLAB or Octave

MATLAB is a commercial numerical computing environment and programming language. MATLAB allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages.

Octave is an open-source Matlab-like package, available on nearly all platforms. Currently Octave relies on Gnuplot for plotting, and has more limited graphics capabilities than GNU R or MATLAB.

12.6.4 Gnuplot

Gnuplot is a very popular command-line program that can generate two- and three-dimensional plots of functions and data. The program runs on all major platforms, and it is well supported.

Gnuplot has an interactive command interface. For example, if you have the data files `foo.csv` and `bar.csv` that contain two values per line (x y; such files can be exported with `scavetool` from vector files), you can plot them in the same graph by typing:

```
plot "foo.csv" with lines, "bar.csv" with lines
```

To adjust the y range, you would type:

```
set yrange [0:1.2]
replot
```

Several commands are available to adjust ranges, plotting style, labels, scaling etc. On Windows, you can copy the resulting graph to the clipboard from the Gnuplot window's system menu, then insert it into the application you are working with.

12.6.5 ROOT

ROOT is an object-oriented data analysis framework, with strong support for plotting and graphics in general. ROOT was developed at CERN, and is distributed under a BSD-like license.

ROOT is based on *CINT*, a "C/C++ interpreter" aimed at processing C/C++ scripts. It is probably harder to get started using ROOT than with either Gnuplot or Grace, but you will find that ROOT provides power and flexibility that would be unattainable with the other two programs.

Curt Brune's page at Stanford (http://www.slac.stanford.edu/čurt/omnet++/) shows examples of what you can achieve using ROOT with OMNeT++.

12.6.6 Grace

An "honorable mention," *Grace* is a powerful GPL data visualization program with a menu-and-dialog graphical user interface for X and Motif. It has also been ported to Windows. Grace is also known as *xmgrace*, and it is a successor of *ACE/gr* or *Xmgr*.

Grace can export graphics in various raster and vector formats, and has many useful features like built-in statistics and analysis functions (e.g. correlation, histogram), fitting, splines, etc., and it also has a built-in programming language.

12.6.7 Spreadsheet Programs

One straightforward solution is to use spreadsheets such as OpenOffice Calc, Microsoft Excel, Gnumeric or Calligra Tables (formerly KSpread). Data can be imported from CSV or other formats, exported with `scavetool` (see 12.5).

Spreadsheets have good charting and statistical features. A useful functionality spreadsheets offer for analyzing scalar files is *PivotTable* (Excel) or *DataPilot* (OpenOffice). The drawback of using spreadsheets is limited automation, leading to tedious and repetitive tasks; also, the number of rows is usually limited to about 32,000..64,000, which can be limiting when working with large vector files.

Chapter 13

Eventlog

13.1 Introduction

The eventlog feature and the related tools are completely new in OMNeT++ 4.0. They aim to help in understanding complex simulation models and to help correctly implement the desired component behaviors. Using these tools you will be able to easily examine every minute detail of the simulation back and forth in terms of simulation time, or step-by-step, focusing on the behavior instead of the statistical results of your model.

The eventlog file is created automatically during a simulation run upon explicit request configurable in the ini file. The resulting file can be viewed in the OMNeT++ IDE using the Sequence Chart and the Eventlog Table or can be processed by the command line Eventlog Tool. These tools support filtering the collected data to allow you to focus on events relevant to what you are looking for. They allow examining causality relationships and provide filtering based on simulation times, event numbers, modules and messages.

The simulation kernel records into the eventlog among others: user level messages, creation and deletion of modules, gates and connections, scheduling of self messages, sending of messages to other modules either through gates or directly, and processing of messages (that is events). Optionally, detailed message data can also be automatically recorded based on a message filter. The result is an eventlog file which contains detailed information of the simulation run and later can be used for various purposes.

NOTE: The eventlog file may become quite large for long running simulations (often hundreds of megabytes, but occasionally several gigabytes), because it contains a lot of information about the run, especially when message detail recording is turned on.

13.2 Configuration

To record an eventlog file during the simulation, insert the following line into the ini file:

```
record-eventlog = true
```

NOTE: Eventlog recording is turned off by default, because creating the eventlog file might significantly decrease the overall simulation performance.

13.2.1 File Name

The simulation kernel will write the eventlog file during the simulation into the file specified by the following ini file configuration entry (showing the default file name pattern here):

```
eventlog-file = ${resultdir}/${configname}-${runnumber}.elog
```

13.2.2 Recording Intervals

The size of an eventlog file is approximately proportional to the number of events it contains. To reduce the file size and speed up the simulation, it might be useful to record only certain events. The `eventlog-recording-intervals` configuration option instructs the kernel to record events only in the specified intervals. The syntax is similar to that of `vector-recording-intervals`.

An example:

```
eventlog-recording-intervals = ..10.2, 22.2..100, 233.3..
```

13.2.3 Recording Modules

Another factor that affects the size of an eventlog file is the number of modules for which the simulation kernel records events during the simulation. The `module-eventlog-recording` per-module configuration option instructs the kernel to record only the events that occurred in the matching modules. The default is to record events from all modules. This configuration option only applies to simple modules.

The following example records events from any of the routers whose index is between 10 and 20, and turns off recording for all other modules.

```
**.router[10..20].**.module-eventlog-recording = true
**.module-eventlog-recording = false
```

13.2.4 Recording Message Data

Since recording message data dramatically increases the size of the eventlog file and also slows down the simulation, it is turned off by default, even if writing the eventlog is enabled. To turn on message data recording, supply a value for the `eventlog-message-detail-pattern` option in the ini file.

An example configuration for an IEEE 80211 model that records the `encapsulationMsg` field and all other fields whose name ends in `Address`, from messages whose class name ends in `Frame` looks like this:

```
eventlog-message-detail-pattern = *Frame:encapsulatedMsg,*Address
```

An example configuration for a TCP/IP model that records the port and address fields in all network packets looks like the following:

```
eventlog-message-detail-pattern =
 PPPFrame:encapsulatedPacket|IPDatagram:encapsulatedPacket,*Address|TCPSegment:*Port
```

13.3 Eventlog Tool

The Eventlog Tool is a command line tool to process eventlog files. Invoking it without parameters will display usage information. The following are the most useful commands for users.

13.3.1 Filter

The eventlog tool provides off line filtering that is usually applied to the eventlog file after the simulation has been finished and before actually opening it in the OMNeT++ IDE or processing it by any other means. Use the filter command and its various options to specify what should be present in the result file.

13.3.2 Echo

Since the eventlog file format is text based and users are encouraged to implement their own filters, a way is needed to check whether an eventlog file is correct. The echo command provides a way to check this and help users creating custom filters. Anything not echoed back by the eventlog tool will not be taken into consideration by the other tools found in the OMNeT++ IDE.

NOTE: Custom filter tools should filter out whole events only, otherwise the consequences are undefined.

Chapter 14

Documenting NED and Messages

14.1 Overview

OMNeT++ provides a tool which can generate HTML documentation from NED files and message definitions. Like Javadoc and Doxygen, the NED documentation tool makes use of source code comments. The generated HTML documentation lists all modules, channels, messages, etc., and presents their details including description, gates, parameters, assignable submodule parameters, and syntax-highlighted source code. The documentation also includes clickable network diagrams (exported from the graphical editor) and usage diagrams as well as inheritance diagrams.

The documentation tool integrates with Doxygen, meaning that it can hyperlink simple modules and message classes to their C++ implementation classes in the Doxygen documentation. If you also generate the C++ documentation with some Doxygen features turned on (such as *inline-sources* and *referenced-by-relation*, combined with *extract-all*, *extract-private* and *extract-static*), the result is an easily browsable and very informative presentation of the source code. Of course, one still has to write documentation comments in the code.

In the 4.0 version, the documentation tool is part of the Eclipse-based simulation IDE.

14.2 Documentation Comments

Documentation is embedded in normal comments. All `//` comments that are in the "right place" (from the documentation tool's point of view) will be included in the generated documentation. [1]

Example:

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//
simple Gen
{
```

[1] In contrast, Javadoc and Doxygen use special comments (those beginning with `/**`, `///`, `//<` or a similar marker) to distinguish documentation from "normal" comments in the source code. In OMNeT++ there is no need for that: NED and the message syntax is so compact that practically all comments one would want to write in them can serve documentation purposes.

```
    parameters:
        string destAddress;  // destination MAC address
        int protocolId;      // value for SSAP/DSAP in Ethernet frame
        double waitMean @unit(s); // mean for exponential interarrival times
    gates:
        output out;  // to Ethernet LLC
}
```

You can also place comments above parameters and gates. This is useful if they need long explanations. Example:

```
//
// Deletes packets and optionally keeps statistics.
//
simple Sink
{
    parameters:
        // You can turn statistics generation on and off. This is
        // a very long comment because it has to be described what
        // statistics are collected (or not).
        bool collectStatistics = default(true);
    gates:
        input in;
}
```

14.2.1 Private Comments

If you want a comment line *not* to appear in the documentation, begin it with `//#`. Those lines will be ignored by the documentation tool, and can be used to make "private" comments like `FIXME` or `TODO`, or to comment out unused code.

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//# TODO above description needs to be refined
//
simple Gen
{
    parameters:
        string destAddress;  // destination MAC address
        int protocolId;      // value for SSAP/DSAP in Ethernet frame
        //# double burstiness;  -- not yet supported
        double waitMean @unit(s); // mean for exponential interarrival times
    gates:
        output out;  // to Ethernet LLC
}
```

14.2.2 More on Comment Placement

Comments should be written where the tool will find them. This is a) immediately above the documented item, or b) after the documented item, on the same line.

In the former case, make sure there is no blank line left between the comment and the documented item. Blank lines detach the comment from the documented item.

Example:

```
// This is wrong! Because of the blank line, this comment is not
// associated with the following simple module!

simple Gen
{
    ...
}
```

Do not try to comment groups of parameters together. The result will be awkward.

14.3 Referring to Other NED and Message Types

You can reference other NED and message types by name in comments. There are two styles in which references can be written: automatic linking and tilde linking. The same style must be following throughout the whole project, and the correct one must be selected in the documentation generator tool when it is run.

14.3.1 Automatic Linking

In the automatic linking style, words that match existing NED of message types are hyperlinked automatically. It is usually enough to write the simple name of the type (e.g. `TCP`), you don't need to spell out the fully qualified type (`inet.transport.tcp.TCP`), although you can.

Automatic hyperlinking is sometimes overly agressive. For example, when you write `IP address` in a comment and an `IP` module exists in the project, it will create a hyperlink to the module, which is probably not what you want. You can prevent hyperlinking of a word by inserting a backslash in front it: `\IP address`. The backslash will not appear in the HTML output. The `<nohtml>` tag will also prevent hyperlinking words in the enclosed text: `<nohtml>IP address</nohtml>`. On the other hand, if you deliberately want to print a backslash immediately in front of a word (e.g. output *"use \t to print a Tab"*), use either two backslashes (`use \\t...`) or the `<nohtml>` tag (`<nohtml>use \t...</nohtml>`). Backslashes in other contexts (i.e. when not in front of a word) do not have a special meaning, and are preserved in the output.

The detailed rules:

1. Words matching a type name are automatically hyperlinked
2. A backslash immediately followed by an identifier (i.e. letter or underscore) prevents hyperlinking, and the backslash is removed from the output
3. A double backslash followed by an identifier produces a single backslash, plus the potentially hyperlinked identifier
4. Backslashes in any other contexts are not interpreted, and preserved in the output
5. Tildes are not interpreted, and are preserved in the output
6. Inside `<nohtml>`, no backslash processing or hyperlinking takes place

14.3.2 Tilde Linking

In the tilde style, only words that are explicitly marked with a tilde are subject to hyperlinking: `~TCP`, `~inet.transport.tcp.TCP`.

To produce a literal tilde followed by an identifier in the output (for example, to output *"the ~TCP() destructor"*), you need to double the tilde character: `the ~~TCP() destructor`.

The detailed rules:

1. Words matching a type name are *not* hyperlinked automatically
2. A tilde immediately followed by an identifier (i.e. letter or underscore) will be hyperlinked, and the tilde is removed from the output. It is considered an error if there is no type with that name.
3. A double tilde followed by an identifier produces a single tilde plus the identifier
4. Tildes in any other contexts are not interpreted, and preserved in the output
5. Backslashes are not interpreted, and are preserved in the output
6. Inside `<nohtml>`, no tilde processing or hyperlinking takes place

14.4 Text Layout and Formatting

14.4.1 Paragraphs and Lists

If you write longer descriptions, you will need text formatting capabilities. Text formatting works like in Javadoc or Doxygen – you can break up the text into paragraphs and create bulleted/numbered lists without special commands, and use HTML for more fancy formatting.

Paragraphs are separated by empty lines, like in LaTeX or Doxygen. Lines beginning with '–' will be turned into bulleted lists, and lines beginning with '`-#`' into numbered lists.

Example:

```
//
// Ethernet MAC layer. MAC performs transmission and reception of frames.
//
// Processing of frames received from higher layers:
// - sends out frame to the network
// - no encapsulation of frames -- this is done by higher layers.
// - can send PAUSE message if requested by higher layers (PAUSE protocol,
//   used in switches). PAUSE is not implemented yet.
//
// Supported frame types:
// -# IEEE 802.3
// -# Ethernet-II
//
```

14.4.2 Special Tags

The documentation tool understands the following tags and will render them accordingly: `@author`, `@date`, `@todo`, `@bug`, `@see`, `@since`, `@warning`, `@version`. Example usage:

```
//
// @author Jack Foo
// @date 2005-02-11
//
```

14.4.3 Text Formatting Using HTML

Common HTML tags are understood as formatting commands. The most useful tags are: `<i>..</i>` (italic), `<b>..</b>` (bold), `<tt>..</tt>` (typewriter font), `<sub>..</sub>` (subscript), `<sup>..</sup>` (superscript), `<br>` (line break), `<h3>` (heading), `<pre>..</pre>` (preformatted text) and `<a href=..>..</a>` (link), as well as a few other tags used for table creation (see below). For example, `<i>Hello</i>` will be rendered as "*Hello*" (using an italic font).

The complete list of HTML tags interpreted by the documentation tool are: `<a>`, `<b>`, `<body>`, `<br>`, `<center>`, `<caption>`, `<code>`, `<dd>`, `<dfn>`, `<dl>`, `<dt>`, `<em>`, `<form>`, `<font>`, `<hr>`, `<h1>`, `<h2>`, `<h3>`, `<i>`, `<input>`, `<img>`, `<li>`, `<meta>`, `<multicol>`, `<ol>`, `<p>`, `<small>`, `<span>`, `<strong>`, `<sub>`, `<sup>`, `<table>`, `<td>`, `<th>`, `<tr>`, `<tt>`, `<kbd>`, `<ul>`, `<var>`.

Any tags not in the above list will not be interpreted as formatting commands but will be printed verbatim – for example, `<what>bar</what>` will be rendered literally as "<what>bar</what>" (unlike HTML where unknown tags are simply ignored, i.e. HTML would display "bar").

If you insert links to external pages (web sites), its useful to add the `target="_blank"` attribute to ensure pages come up in a new browser window and not just in the current frame which looks awkward. (Alternatively, you can use the `target="_top"` attribute which replaces all frames in the current browser).

Examples:

```
//
// For more info on Ethernet and other LAN standards, see the
// <a href="http://www.ieee802.org/" target="_blank">IEEE 802
// Committee's site</a>.
//
```

You can also use the `<a href=..>` tag to create links within the page:

```
//
// See the <a href="#resources">resources</a> in this page.
// ...
// <a name="resources"><b>Resources</b></a>
// ...
//
```

You can use the `<pre>..</pre>` HTML tag to insert source code examples into the documentation. Line breaks and indentation will be preserved, but HTML tags continue to be interpreted (or you can turn them off with `<nohtml>`, see later).

Example:

```
// <pre>
// // my preferred way of indentation in C/C++ is this:
// <b>for</b> (<b>int</b> i=0; i<10; i++)
// {
//     printf(<i>"%d\n"</i>, i);
```

```
// }
// </pre>
```

will be rendered as

```
// my preferred way of indentation in C/C++ is this:
for (int i=0; i<10; i++)
{
    printf("%d\n", i);
}
```

HTML is also the way to create tables. The example below

```
//
// <table border="1">
//   <tr>  <th>#</th> <th>number</th> </tr>
//   <tr>  <td>1</td> <td>one</td>    </tr>
//   <tr>  <td>2</td> <td>two</td>    </tr>
//   <tr>  <td>3</td> <td>three</td>  </tr>
// </table>
//
```

will be rendered approximately as:

#	number
1	one
2	two
3	three

14.4.4 Escaping HTML Tags

Sometimes you may need to turn off interpreting HTML tags (`<i>`, `<b>`, etc.) as formatting instructions, and rather you want them to appear as literal `<i>`, `<b>` text in the documentation. You can achieve this via surrounding the text with the `<nohtml>...</nohtml>` tag. For example,

```
// Use the <nohtml><i></nohtml> tag (like <tt><nohtml><i>this</i></nohtml><tt>)
// to write in <i>italic</i>.
```

will be rendered as "Use the **<i>** tag (like `<i>this</i>`) to write in *italic*."

`<nohtml>...</nohtml>` will also prevent `opp_neddoc` from hyperlinking words that are accidentally the same as an existing module or message name. Prefixing the word with a backslash will achieve the same. That is, either of the following will do:

```
// In <nohtml>IP</nohtml> networks, routing is...
```

```
// In \IP networks, routing is...
```

Both will prevent hyperlinking the word *IP* if you happen to have an `IP` module in the NED files.

14.5 Customizing and Adding Pages

14.5.1 Adding a Custom Title Page

The title page is the one that appears in the main frame after opening the documentation in the browser. By default it contains a boilerplate text with the generic title *"OMNeT++ Model Documentation"*. You probably want to customize that, and at least change the title to the name of the documented simulation model.

You can supply your own version of the title page adding a `@titlepage` directive to a file-level comment (a comment that appears at the top of a NED file, but is separated from the first `import`, `channel`, `module`, etc. definition by at least one blank line). In theory you can place your title page definition into any NED or MSG file, but it is probably a good idea to create a separate `package.ned` file for it.

The lines you write after the `@titlepage` line up to the next `@page` line (see later) or the end of the comment will be used as the title page. You probably want to begin with a title because the documentation tool doesn't add one (it lets you have full control over the page contents). You can use the `<h1>..</h1>` HTML tag to define a title.

Example:

```
//
// @titlepage
// <h1>Ethernet Model Documentation</h1>
//
// This documents the Ethernet model created by David Wu and refined by Andras
// Varga at CTIE, Monash University, Melbourne, Australia.
//
```

14.5.2 Adding Extra Pages

You can add new pages to the documentation in a similar way as customizing the title page. The directive to be used is `@page`, and it can appear in any file-level comment (see above).

The syntax of the `@page` directive is the following:

```
// @page filename.html, Title of the Page
```

Choose a file name that doesn't collide with the files generated by the documentation tool (such as `index.html`). If the file name does not end in `.html` already, it will be appended. The page title you supply will appear on the top of the page as well as in the page index.

The lines after the `@page` line up to the next `@page` line or the end of the comment will be used as the page body. You don't need to add a title because the documentation tool automatically adds one.

Example:

```
//
// @page structure.html, Directory Structure
//
// The model core model files and the examples have been placed
// into different directories. The <tt>examples/</tt> directory...
//
//
```

```
// @page examples.html, Examples
// ...
//
```

You can create links to the generated pages using standard HTML, using the `<a href="...">...</a>` tag. All HTML files are placed in a single directory, so you don't have to worry about specifying directories.

Example:

```
//
// @titlepage
// ...
// The structure of the model is described <a href="structure.html">here</a>.
//
```

14.5.3 Incorporating Externally Created Pages

You may want to create pages outside the documentation tool (e.g. using a HTML editor) and include them in the documentation. This is possible, all you have to do is declare such pages with the `@externalpage` directive in any of the NED files, and they will be added to the page index. The pages can then be linked to from other pages using the HTML `<a href="...">...</a>` tag.

The `@externalpage` directive is similar in syntax to `@page`:

```
// @externalpage filename.html, Title of the Page
```

The documentation tool does not check if the page exists or not. It is your responsibility to copy it manually into the directory of the generated documentation, and to make sure the hyperlink works.

14.6 File inclusion

You can include content into the documentation comment with the `@include` directive. It expects the path of the file to be incuded relative to the file that includes it.

The line of the `@include` directive will be replaced by the content of the file. The lines of the included file do not need to start with `//`, but otherwise they are processed in the same way as the NED comments. They can include other files, but circular includes are not allowed.

```
// ...
// @include ../copyright.txt
// ...
```

Chapter 15

Parallel Distributed Simulation

15.1 Introduction to Parallel Discrete Event Simulation

OMNeT++ supports parallel execution of large simulations. This section provides a brief picture of the problems and methods of parallel discrete event simulation (PDES). Interested readers are strongly encouraged to look into the literature.

For parallel execution, the model is to be partitioned into several LPs (logical processes) that will be simulated independently on different hosts or processors. Each LP will have its own local Future Event Set, and thus will maintain its own local simulation time. The main issue with parallel simulations is keeping LPs synchronized in order to avoid violating the causality of events. Without synchronization, a message sent by one LP could arrive in another LP when the simulation time in the receiving LP has already passed the timestamp (arrival time) of the message. This would break causality of events in the receiving LP.

There are two broad categories of parallel simulation algorithms that differ in the way they handle causality problems outlined above:

1. **Conservative algorithms** prevents incausalities from happening. The Null Message Algorithm exploits knowledge of the time when LPs send messages to other LPs, and uses 'null' messages to propagate this information to other LPs. If an LP knows it won't receive any messages from other LPs until $t + \Delta t$ simulation time, it may advance until $t + \Delta t$ without the need for external synchronization. Conservative simulation tends to converge to sequential simulation (slowed down by communication between LPs) if there is not enough parallelism in the model, or parallelism is not exploited by sending a sufficient number of 'null' messages.

2. **Optimistic synchronization** allows incausalities to occur, but detects and repairs them. Repairing involves rollbacks to a previous state, sending out anti-messages to cancel messages sent out during the period that is being rolled back, etc. Optimistic synchronization is extremely difficult to implement, because it requires periodic state saving and the ability to restore previous states. In any case, implementing optimistic synchronization in 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.

15.2 Assessing Available Parallelism in a Simulation Model

OMNeT++ currently supports conservative synchronization via the classic Chandy-Misra-Bryant (or null message) algorithm [CM79]. To assess how efficiently a simulation can be parallelized with this algorithm, we'll need the following variables:

- P *performance* represents the number of events processed per second (ev/sec). [1] P depends on the performance of the hardware and the computation-intensiveness of processing an event. P is independent of the size of the model. Depending on the nature of the simulation model and the performance of the computer, P is usually in the range of 20,000..500,000 ev/sec.

- E *event density* is the number of events that occur per simulated second (ev/simsec). E depends on the model only, and not where the model is executed. E is determined by the size, the detail level and also the nature of the simulated system (e.g. cell-level ATM models produce higher E values than call center simulations.)

- R *relative speed* measures the simulation time advancement per second (simsec/sec). R strongly depends on both the model and on the software/hardware environment where the model executes. Note that $R = P/E$.

- L *lookahead* is measured in simulated seconds (simsec). When simulating telecommunication networks and using link delays as lookahead, L is typically in the msimsec-μsimsec range.

- τ *latency* (sec) characterizes the parallel simulation hardware. τ is the latency of sending a message from one LP to another. τ can be determined using simple benchmark programs. The authors' measurements on a Linux cluster interconnected via a 100Mb Ethernet switch using MPI yielded τ=22μs which is consistent with measurements reported in [OF00]. Specialized hardware such as Quadrics Interconnect [Qua] can provide τ=5μs or better.

In large simulation models, P, E and R usually stay relatively constant (that is, display little fluctuations in time). They are also intuitive and easy to measure. The OMNeT++ displays these values on the GUI while the simulation is running, see Figure 15.1. Cmdenv can also be configured to display these values.

Figure 15.1: Performance bar in OMNeT++ showing P, R and E

After having approximate values of P, E, L and τ, calculate the λ *coupling factor* as the ratio of LE and τP:

$\lambda = (LE)/(\tau P)$

Without going into the details: if the resulting λ value is at minimum larger than one, but rather in the range 10..100, there is a good chance that the simulation will perform well when run in parallel. With $\lambda < 1$, poor performance is guaranteed. For details see the paper [VŞE03].

[1] Notations: *ev:* events, *sec:* real seconds, *simsec:* simulated seconds

15.3 Parallel Distributed Simulation Support in OMNeT++

15.3.1 Overview

This chapter presents the parallel simulation architecture of OMNeT++. The design allows simulation models to be run in parallel without code modification – it only requires configuration. The implementation relies on the approach of placeholder modules and proxy gates to instantiate the model on different LPs – the placeholder approach allows simulation techniques such as topology discovery and direct message sending to work unmodified with PDES. The architecture is modular and extensible, so it can serve as a framework for research on parallel simulation.

The OMNeT++ design places a big emphasis on *separation of models from experiments*. The main rationale is that usually a large number of simulation experiments need to be done on a single model before a conclusion can be drawn about the real system. Experiments tend to be ad-hoc and change much faster than simulation models; thus it is a natural requirement to be able to carry out experiments without disturbing the simulation model itself.

Following the above principle, OMNeT++ allows simulation models to be executed in parallel without modification. No special instrumentation of the source code or the topology description is needed, as partitioning and other PDES configuration is entirely described in the configuration files.

OMNeT++ supports the Null Message Algorithm with static topologies, using link delays as lookahead. The laziness of null message sending can be tuned. Also supported is the Ideal Simulation Protocol (ISP) introduced by Bagrodia in 2000 [BT00]. ISP is a powerful research vehicle to measure the efficiency of PDES algorithms, both optimistic and conservative; more precisely, it helps determine the maximum speedup achievable by any PDES algorithm for a particular model and simulation environment. In OMNeT++, ISP can be used for benchmarking the performance of the Null Message Algorithm. Additionally, models can be executed without any synchronization, which can be useful for educational purposes (to demonstrate the need for synchronization) or for simple testing.

For the communication between LPs (logical processes), OMNeT++ primarily uses MPI, the Message Passing Interface standard [For94]. An alternative communication mechanism is based on named pipes, for use on shared memory multiprocessors without the need to install MPI. Additionally, a file system based communication mechanism is also available. It communicates via text files created in a shared directory, and can be useful for educational purposes (to analyse or demonstrate messaging in PDES algorithms) or to debug PDES algorithms. Implementation of a shared memory-based communication mechanism is also planned for the future, to fully exploit the power of multiprocessors without the overhead of and the need to install MPI.

Nearly every model can be run in parallel. The constraints are the following:

- modules may communicate via sending messages only (no direct method call or member access) unless mapped to the same processor
- no global variables
- there are some limitations on direct sending (no sending to a *sub*module of another module, unless mapped to the same processor)
- lookahead must be present in the form of link delays
- currently static topologies are supported (we are working on a research project that aims to eliminate this limitation)

PDES support in OMNeT++ follows a modular and extensible architecture. New communication mechanisms can be added by implementing a compact API (expressed as a C++ class) and registering the implementation – after that, the new communications mechanism can be selected for use in the configuration.

New PDES synchronization algorithms can be added in a similar way. PDES algorithms are also represented by C++ classes that have to implement a very small API to integrate with the simulation kernel. Setting up the model on various LPs as well as relaying model messages across LPs is already taken care of and not something the implementation of the synchronization algorithm needs to worry about (although it can intervene if needed, because the necessary hooks are provided).

The implementation of the Null Message Algorithm is also modular in itself in that the lookahead discovery can be plugged in via a defined API. Currently implemented lookahead discovery uses link delays, but it is possible to implement more sophisticated approaches and select them in the configuration.

15.3.2 Parallel Simulation Example

We will use the Parallel CQN example simulation for demonstrating the PDES capabilities of OMNeT++. The model consists of N tandem queues where each tandem consists of a switch and k single-server queues with exponential service times (Figure 15.2). The last queues are looped back to their switches. Each switch randomly chooses the first queue of one of the tandems as destination, using uniform distribution. The queues and switches are connected with links that have nonzero propagation delays. Our OMNeT++ model for CQN wraps tandems into compound modules.

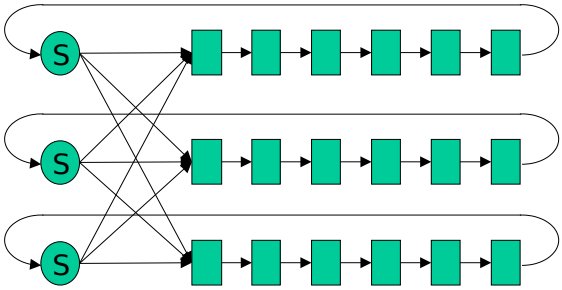


Figure 15.2: The Closed Queueing Network (CQN) model

To run the model in parallel, we assign tandems to different LPs (Figure 15.3). Lookahead is provided by delays on the marked links.

To run the CQN model in parallel, we have to configure it for parallel execution. In OMNeT++, the configuration is in the `omnetpp.ini` file. For configuration, first we have to specify partitioning, that is, assign modules to processors. This is done by the following lines:

```
[General]
*.tandemQueue[0]**.partition-id = 0
*.tandemQueue[1]**.partition-id = 1
*.tandemQueue[2]**.partition-id = 2
```

The numbers after the equal sign identify the LP.

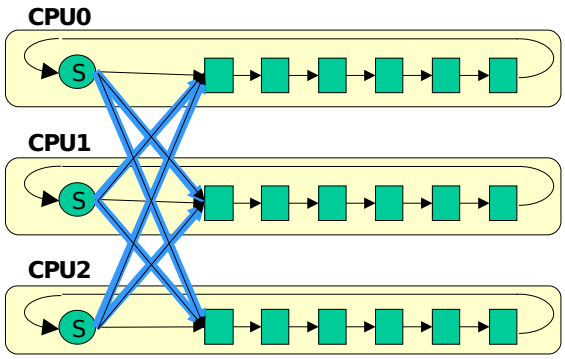


Figure 15.3: Partitioning the CQN model

Then we have to select the communication library and the parallel simulation algorithm, and enable parallel simulation:

```
[General]
parallel-simulation = true
parsim-communications-class = "cMPICommunications"
parsim-synchronization-class = "cNullMessageProtocol"
```

When the parallel simulation is run, LPs are represented by multiple running instances of the same program. When using LAM-MPI [LAM], the mpirun program (part of LAM-MPI) is used to launch the program on the desired processors. When named pipes or file communications is selected, the opp_prun OMNeT++ utility can be used to start the processes. Alternatively, one can run the processes by hand (the -p flag tells OMNeT++ the index of the given LP and the total number of LPs):

```
./cqn -p0,3 &
./cqn -p1,3 &
./cqn -p2,3 &
```

For PDES, one will usually want to select the command-line user interface, and redirect the output to files. (OMNeT++ provides the necessary configuration options.)

The graphical user interface of OMNeT++ can also be used (as evidenced by Figure 15.4), independently of the selected communication mechanism. The GUI interface can be useful for educational or demonstration purposes. OMNeT++ displays debugging output about the Null Message Algorithm, EITs and EOTs can be inspected, etc.

15.3.3 Placeholder Modules, Proxy Gates

When setting up a model partitioned to several LPs, OMNeT++ uses placeholder modules and proxy gates. In the local LP, placeholders represent sibling submodules that are instantiated on other LPs. With placeholder modules, every module has all of its siblings present in the local LP – either as placeholder or as the "real thing". Proxy gates take care of forwarding messages to the LP where the module is instantiated (see Figure 15.5).

The main advantage of using placeholders is that algorithms such as topology discovery embedded in the model can be used with PDES unmodified. Also, modules can use direct message sending to any sibling module, including placeholders. This is so because the destination

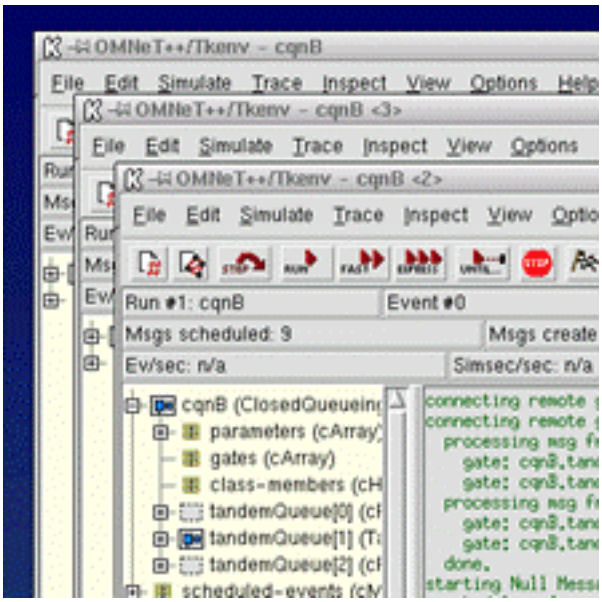


Figure 15.4: Screenshot of CQN running in three LPs

of direct message sending is an input gate of the destination module – if the destination module is a placeholder, the input gate will be a proxy gate which transparently forwards the messages to the LP where the "real" module was instantiated. A limitation is that the destination of direct message sending cannot be a *submodule* of a sibling (which is probably a bad practice anyway, as it violates encapsulation), simply because placeholders are empty and so its submodules are not present in the local LP.

Instantiation of compound modules is slightly more complicated. Since submodules can be on different LPs, the compound module may not be "fully present" on any given LP, and it may have to be present on several LPs (wherever it has submodules instantiated). Thus, compound modules are instantiated wherever they have at least one submodule instantiated, and are represented by placeholders everywhere else (Figure 15.6).

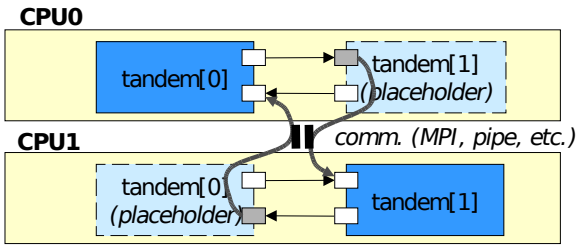


Figure 15.5: Placeholder modules and proxy gates

15.3.4 Configuration

Parallel simulation configuration is the `[General]` section of `omnetpp.ini`.

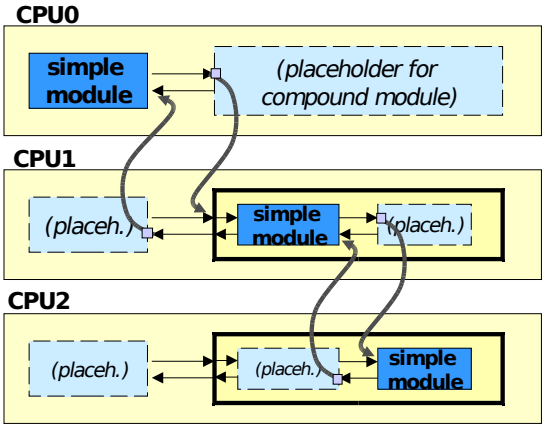


Figure 15.6: Instantiating compound modules

The parallel distributed simulation feature can be turned on with the **parallel-simulation** boolean option.

The **parsim-communications-class** selects the class that implements communication between partitions. The class must implement the `cParsimCommunications` interface.

The **parsim-synchronization-class** selects the parallel simulation algorithm. The class must implement the `cParsimSynchronizer` interface.

The following two options configure the Null Message Algorithm, so they are only effective if `cNullMessageProtocol` has been selected as synchronization class:

- **parsim-nullmessageprotocol-lookahead-class** selects the lookahead class for the NMA; the class must be subclassed from `cNMPLookahead`. The default class is `cLinkDelayLookahead`.

- **parsim-nullmessageprotocol-laziness** expects a number in the $(0,1)$ interval (the default is 0.5), and it ontrols how often NMA should send out null messages; the value is understood in proportion to the lookahead, e.g. 0.5 means every $lookahead/2$ simsec.

The **parsim-debug** boolean option enables/disables printing log messages about the parallel simulation algorithm. It is turned on by default, but for production runs we recommend turning it off.

Other configuration options configure MPI buffer sizes and other details; see options that begin with `parsim-` in Appendix G.

When you are using cross-mounted home directories (the simulation's directory is on a disk mounted on all nodes of the cluster), a useful configuration setting is

```
[General]
fname-append-host = true
```

It will cause the host names to be appended to the names of all output vector files, so that partitions don't overwrite each other's output files. (See section 11.5.3)

15.3.5 Design of PDES Support in OMNeT++

The design of PDES support in OMNeT++ follows a layered approach, with a modular and extensible architecture. The overall architecture is depicted in Figure 15.7.

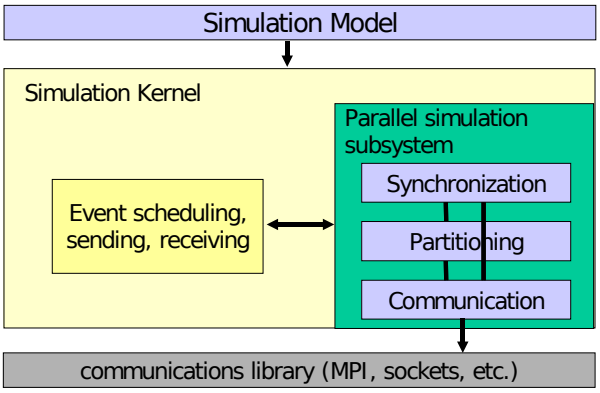


Figure 15.7: Architecture of OMNeT++ PDES implementation

The parallel simulation subsytem is an optional component itself, which can be removed from the simulation kernel if not needed. It consists of three layers, from the bottom up: Communications Layer, Partitioning Layer and Synchronization Layer.

The Communications Layer

The purpose of the Communications Layer is to provide elementary messaging services between partitions for the upper layer. The services include send, blocking receive, nonblocking receive and broadcast. The send/receive operations work with *buffers*, which encapsulate packing and unpacking operations for primitive C++ types. The message class and other classes in the simulation library can pack and unpack themselves into such buffers. The Communications layer API is defined in the `cParsimCommunications` interface (abstract class); specific implementations like the MPI one (`cMPICommunications`) subclass from this, and encapsulate MPI send/receive calls. The matching buffer class `cMPICommBuffer` encapsulates MPI pack/unpack operations.

The Partitioning Layer

The Partitioning Layer is responsible for instantiating modules on different LPs according to the partitioning specified in the configuration, for configuring proxy gates. During the simulation, this layer also ensures that cross-partition simulation messages reach their destinations. It intercepts messages that arrive at proxy gates and transmits them to the destination LP using the services of the Communications Layer. The receiving LP unpacks the message and injects it at the gate the proxy gate points at. The implementation basically encapsulates the `cParsimSegment`, `cPlaceholderModule`, `cProxyGate` classes.

The Synchronization Layer

The Synchronization Layer encapsulates the parallel simulation algorithm. Parallel simulation algorithms are also represented by classes, subclassed from the `cParsimSynchronizer`

abstract class. The parallel simulation algorithm is invoked on the following hooks: event scheduling, processing model messages outgoing from the LP, and messages (model messages or internal messages) arriving from other LPs. The first hook, event scheduling, is a function invoked by the simulation kernel to determine the next simulation event; it also has full access to the future event set (FES) and can add/remove events for its own use. Conservative parallel simulation algorithms will use this hook to block the simulation if the next event is unsafe, e.g. the null message algorithm implementation (`cNullMessageProtocol`) blocks the simulation if an EIT has been reached until a null message arrives (see [BT00] for terminology); also it uses this hook to periodically send null messages. The second hook is invoked when a model message is sent to another LP; the null message algorithm uses this hook to piggyback null messages on outgoing model messages. The third hook is invoked when any message arrives from other LPs, and it allows the parallel simulation algorithm to process its own internal messages from other partitions; the null message algorithm processes incoming null messages here.

The Null Message Protocol implementation itself is modular; it employs a separate, configurable lookahead discovery object. Currently only link delay based lookahead discovery has been implemented, but it is possible to implement more sophisticated types.

The Ideal Simulation Protocol (ISP; see [BT00]) implementation consists of two parallel simulation protocol implementations: the first one is based on the null message algorithm and additionally records the external events (events received from other LPs) to a trace file; the second one executes the simulation using the trace file to find out which events are safe and which are not.

Note that although we implemented a conservative protocol, the provided API itself would allow implementing optimistic protocols, too. The parallel simulation algorithm has access to the executing simulation model, so it could perform saving/restoring model state if model objects support this [2].

We also expect that because of the modularity, extensibility and clean internal architecture of the parallel simulation subsystem, the OMNeT++ framework has the potential to become a preferred platform for PDES research.

[2] Unfortunately, support for state saving/restoration needs to be individually and manually added to each class in the simulation, including user-programmed simple modules.

Chapter 16

Plug-in Extensions

16.1 Overview

OMNeT++ is an open system, and several details of its operation can be customized via plug-ins. To create a plug-in, you generally need to write a C++ class that implements a certain interface (i.e. subclasses from a C++ abstract class), and register it in OMNeT++. The plug-in class can be activated for a particular simulation with a corresponding configuration option.

The following plug-in interfaces are supported:

- `cRNG`. Interface for random number generators.
- `cScheduler`. The scheduler class. This plug-in interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation.
- `cConfigurationEx`. Configuration provider plug-in. This plug-in interface lets you replace `omnetpp.ini` with some other implementation, for example a database.
- `cOutputScalarManager`. It handles recording the scalar output data. The default output scalar manager is `cFileOutputScalarManager`, defined in the Envir library.
- `cOutputVectorManager`. It handles recording the output from `cOutVector` objects. The default output vector manager is `cIndexedFileOutputVectorManager`, defined in the Envir library.
- `cSnapshotManager`. It provides an output stream to which snapshots are written (see section 7.10.5). The default snapshot manager is `cFileSnapshotManager`, defined in the Envir library.

The classes (`cRNG`, `cScheduler`, etc.) are documented in the API Reference.

To actually implement and select a plug-in for use:

1. Subclass the given interface class (e.g. for a custom RNG, `cRNG`) to create your own version.
2. Register the class by putting the `Register_Class(MyRNGClass)` line into the C++ source.

3. Compile and link your interface class into the OMNeT++ simulation executable. IMPORTANT: make sure the executable actually contains the code of your class! Over-optimizing linkers (esp. on Unix) tend to leave out code to which there seem to be no external reference.

4. Add an entry to `omnetpp.ini` to tell Envir use your class instead of the default class. For RNGs, this setting is `rng-class` in the `[General]` section.

16.2 Plug-in Descriptions

16.2.1 Defining a New Random Number Generator

The new RNG C++ class must implement the `cRNG` interface, and can be activated with the **`rng-class`** configuration option.

16.2.2 Defining a New Scheduler

The scheduler plug-in interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation.

The scheduler C++ class must implement the `cScheduler` interface, and can be activated with the **`scheduler-class`** configuration option.

To see examples of scheduler classes, check the `cRealTimeScheduler` class in the simulation kernel, and `cSocketRTScheduler` which is part of the *Sockets* sample simulation.

16.2.3 Defining a New Configuration Provider

Overview

The configuration provider plug-in lets you replace ini files with some other storage implementation, for example a database. The configuration provider C++ class must implement the `cConfigurationEx` interface, and can be activated with the **`configuration-class`** configuration option.

The `cConfigurationEx` interface abstracts the inifile-based data model to some degree. It assumes that the configuration data consists of several *named configurations*. Before every simulation run, one of the *named configurations* is activated, and from then on, all queries into the configuration operate on the *active named configuration* only.

It practice, you will probably use the `SectionBasedConfiguration` class (in `src/envir`) or subclass from it, because it already implements a lot of functionality that you would otherwise have to.

`SectionBasedConfiguration` does not assume ini files or any other particular storage format; instead, it accepts an object that implements the `cConfigurationReader` interface to provide the data in raw form to it. The default implementation of `cConfigurationReader` is `InifileReader`.

The Startup Sequence

From the configuration plug-in's point of view, the startup sequence looks like the following (see `src/envir/startup.cc` in the source code):

1. First, ini files specified on the command-line are read into a *boot-time configuration object*. The boot-time configuration is always a `SectionBasedConfiguration` with `InifileReader`.
2. Shared libraries are loaded (see the `-l` command-line option, and the `load-libs` configuration option). This allows configuration classes to come from shared libraries.
3. The `configuration-class` configuration option is examined. If it is present, a configuration object of the given class is instantiated, and replaces the boot-time configuration. The new configuration object is initialized from the boot-time configuration, so that it can read parameters (e.g. database connection parameters, XML file name, etc) from it. Then the boot-time configuration object is deallocated.
4. The `load-libs` option from the new configuration object is processed.
5. Then everything goes on as normally, using the new configuration object.

Providing a Custom Configuration Class

To replace the configuration object with your custom implementation, you would write the class:

```
#include "cconfiguration.h"

class CustomConfiguration : public cConfigurationEx
{
   ...
};

Register_Class(CustomConfiguration);
```

and then activate it in the boot-time configuration:

```
[General]
configuration-class = CustomConfiguration
```

Providing a Custom Reader for SectionBasedConfiguration

As said already, writing a configuration class from scratch can be a lot of work, and it may be more practical to reuse `SectionBasedConfiguration` with a different configuration reader class. This can be done with `sectionbasedconfig-configreader-class` config option, interpreted by `SectionBasedConfiguration`. Specify the following in your boot-time ini file:

```
[General]
configuration-class = SectionBasedConfiguration
sectionbasedconfig-configreader-class = <my new reader class>
```

The configuration reader class should look like this:

```
#include "cconfigreader.h"

class DatabaseConfigurationReader : public cConfigurationReader
{
   ...
};

Register_Class(DatabaseConfigurationReader);
```

16.2.4 Defining a New Output Scalar Manager

`cOutputScalarManager` handles recording the scalar output data. The default output scalar manager is `cFileOutputScalarManager`, defined in the Envir library.

The new class can be activated with the **`outputscalarmanager-class`** configuration option.

16.2.5 Defining a New Output Vector Manager

`cOutputVectorManager` handles recording the output from `cOutVector` objects. The default output vector manager is `cIndexedFileOutputVectorManager`, defined in the Envir library.

The new class can be activated with the **`outputvectormanager-class`** configuration option.

16.2.6 Defining a New Snapshot Manager

`cSnapshotManager` provides an output stream to which snapshots are written (see section 7.10.5). The default snapshot manager is `cFileSnapshotManager`, defined in the Envir library.

The new class can be activated with the **`snapshotmanager-class`** configuration option.

16.3 Accessing the Configuration

16.3.1 Defining New Configuration Options

New configuration options need to be declared with one of the appropriate registration macros. These macros are:

```
Register_GlobalConfigOption(ID, NAME, TYPE, DEFAULTVALUE, DESCRIPTION)
Register_PerRunConfigOption(ID, NAME, TYPE, DEFAULTVALUE, DESCRIPTION)
Register_GlobalConfigOptionU(ID, NAME, UNIT, DEFAULTVALUE, DESCRIPTION)
Register_PerRunConfigOptionU(ID, NAME, UNIT, DEFAULTVALUE, DESCRIPTION)
Register_PerObjectConfigOption(ID, NAME, TYPE, DEFAULTVALUE, DESCRIPTION)
Register_PerObjectConfigOptionU(ID, NAME, UNIT, DEFAULTVALUE, DESCRIPTION)
```

Config options come in three flavors, as indicated by the macro names:

- *Global* options affect all configurations (i.e. they are only accepted in the `[General]` section but not in `[Config <name>]` sections)

- *Per-Run* options can be specified in any section (i.e. both in `[General]` and in `[Config <name>]` sections). They are specific to a particular section. Their names always contain a hyphen (-) character.
- *Per-Object* options can be specified in any section (i.e. both in `[General]` and in `[Config <name>]` sections). They are specific to an object or group of objects and their name must not contain a hypen (-) character.

The macro arguments are as follows:

- *ID* is a C++ identifier that will let you refer to the configuration option in *cConfiguration* member functions. (It is actually pointer to a `cConfigOption` object that the macro creates.)
- *NAME* is the name of the option (a string).
- *TYPE* is the data type of the option; it must be one of: `CFG_BOOL`, `CFG_INT`, `CFG_DOUBLE`, `CFG_STRING`, `CFG_FILENAME`, `CFG_FILENAMES`, `CFG_PATH`, `CFG_CUSTOM`. The most significant difference between filesystem-related types (filename, filenames, path) and plain strings is that relative filenames and paths are automatically converted to absolute when the configuration is read, with the base directory being the location of the ini file where the configuration entry was read from.
- *UNIT* is a string that names the measurement unit in which the option's value is to be interpreted; it implies type `CFG_DOUBLE`.
- *DEFAULTVALUE* is the default value in textual form (string); this should be `NULL` if the option has no default value.
- *DESCRIPTION* is an arbitrarily long string that describes the purpose and and operation of the option. It will be used in help texts etc.

For example, the `debug-on-errors` macro is declared in the following way:

```
Register_GlobalConfigOption(CFGID_DEBUG_ON_ERRORS,
                            "debug-on-errors", CFG_BOOL, "false",
                            "When enabled, runtime errors will etc etc...");
```

NOTE: Registration is necessary because from the 4.0 version, OMNeT++ validates the configuration on startup, in order to be able to report invalid or mistyped option names and other errors.

16.3.2 Reading Values from the Configuration

The configuration is accessible via the `getConfig()` method of `cEnvir`. It returns a pointer to the configuration object (`cConfiguration`):

```
cConfiguration *config = ev.getConfig();
```

`cConfiguration` provides several methods for querying the configuration.

NOTE: The configuration object provides a flattened view of the ini file. Sections inheriting from each other are merged for you. Configuration options provided on the command line in the form `-option=value` are added first to the object. This ensures that the command line options take precedence over the values specified in the INI file.

```
const char *getAsCustom(cConfigOption *entry, const char *fallbackValue=NULL);
bool getAsBool(cConfigOption *entry, bool fallbackValue=false);
long getAsInt(cConfigOption *entry, long fallbackValue=0);
double getAsDouble(cConfigOption *entry, double fallbackValue=0);
std::string getAsString(cConfigOption *entry, const char *fallbackValue="");
std::string getAsFilename(cConfigOption *entry);
std::vector<std::string> getAsFilenames(cConfigOption *entry);
std::string getAsPath(cConfigOption *entry);
```

fallbackValue is returned if the value is not specified in the configuration, and there is no default value.

```
bool debug = ev.getConfig()->getAsBool(CFGID_PARSIM_DEBUG);
```

16.4 Implementing a New User Interface

It is possible to extend OMNeT++ with a new user interface. The new user interface will have fully equal rights to Cmdenv and Tkenv; that is, it can be activated by starting the simulation executable with the `-u` *<name>* command-line or the `user-interface` configuration option, it can be made the default user interface, it can define new command-line options and configuration options, and so on.

User interfaces must implement (i.e. subclass from) `cRunnableEnvir`, and must be registered to OMNeT++ with the `Register_OmnetApp()` macro. In practice, you will almost always want to subclass `EnvirBase` instead of `cRunnableEnvir`, because `EnvirBase` already implements lots of functionality that otherwise you'd have to.

> **NOTE:** If you want something completely different from what `EnvirBase` provides, such as embedding the simulation kernel into another application, then you should be reading section 17.2, not this one.

An example user interface:

```
#include "envirbase.h"

class FooEnv : public EnvirBase
{
    ...
};

Register_OmnetApp("FooEnv", FooEnv, 30, "an experimental user interface");
```

The `envirbase.h` header comes from the `src/envir` directory, so it is necessary to add it to the include path (`-I`).

The arguments to `Register_OmnetApp()` include the user interface name (for use with the `-u` and **`user-interface`** options), the C++ class that implements it, a weight for default user interface selection (if `-u` is missing, the user interface with the largest weight will be activated), and a description string (for help and other purposes).

The C++ class should implement all methods left pure virtual in `EnvirBase`, and possibly others if you want to customize their behavior. One method that you will surely want to reimplement is `run()` – this is where your user interface runs. When this method exits, the simulation program exits.

NOTE: A good starting point for implementing your own user interface is Cmdenv – just copy and modify its source code to quickly get going.

Chapter 17

Embedding the Simulation Kernel

17.1 Architecture

OMNeT++ has a modular architecture. The following diagram illustrates the high-level architecture of the OMNeT++ simulations:

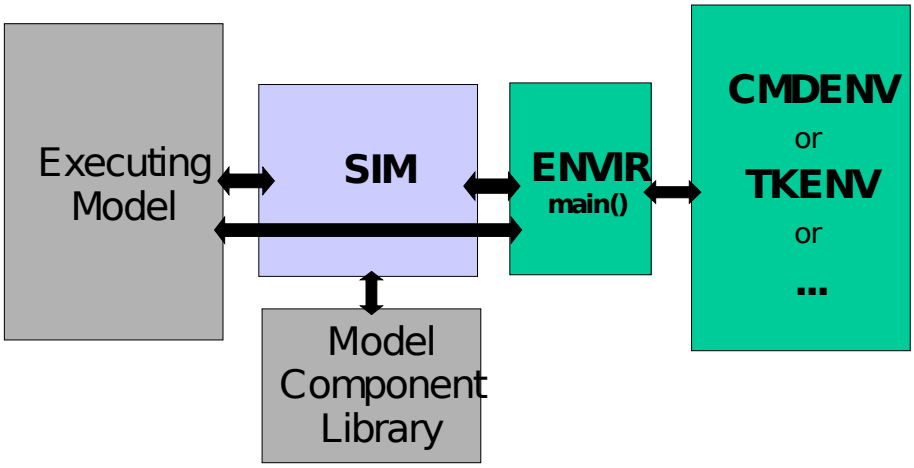


Figure 17.1: Architecture of OMNeT++ simulation programs

The rectangles in the picture represent the following components:

- **Sim** is the simulation kernel and class library. Sim is a library linked to your simulation program.
- **Envir** is another library containing all the code that is common to all the user interfaces. `main()` also exists in the Envir library. 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 the Envir library can be customized using plugin interfaces. Embedding OMNeT++ into applications can be achieved by implementing a new user interface in addition to Cmdenv and Tkenv, or by replacing Envir with another implementation of `ev` (see sections 16.4 and 17.2.)

- **Cmdenv and Tkenv** are specific user interface implementations. The simulation is linked either to the Cmdenv or Tkenv user interfaces, or to both.
- The **Model Component Library** includes simple module definitions and their C++ implementations, compound module types, channels, networks, message types, and everything belonging to models that have been linked to the simulation program. A simulation program can run any model that contains all of the required linked components.
- The **Executing Model** is the model that is set up for simulation. This model contains objects (modules, channels, and so on) that are all instances of the components in the model component library.

The arrows in the figure describe how components interact with each other:

- **Executing Model** ⇔ **Sim**. The simulation kernel manages the future events and activates modules in the executing model as events occur. The modules of the executing model are stored in the main object of Sim, `simulation` (of class `cSimulation`). In turn, the executing model calls functions in the simulation kernel and uses classes in the Sim library.
- **Sim** ⇔ **Model Component Library**. The simulation kernel instantiates simple modules and other components when the simulation model is set up at the beginning of the simulation run. In addition, it refers to the component library when dynamic module creation is used. The mechanisms for registering and looking up components in the model component library are implemented as part of Sim.
- **Executing Model** ⇔ **Envir**. The `ev` object, logically part of Envir, is the facade of the user interface towards the executing model. The model uses `ev` to write debug logs (`ev«`, `ev.printf()`).
- **Sim** ⇔ **Envir**. Envir is in full command of what happens in the simulation program. Envir contains the `main()` function where execution begins. Envir determines which models should be set up for simulation, and instructs Sim to do so. Envir contains the main simulation loop (*determine-next-event*, *execute-event* sequence) and invokes the simulation kernel for the necessary functionality (event scheduling and event execution are implemented in Sim). Envir catches and handles errors and exceptions that occur in the simulation kernel or in the library classes during execution. Envir presents a single facade object (`ev`) that represents the environment (user interface) toward Sim – no Envir internals are visible to Sim or the executing model. During simulation model setup, Envir supplies parameter values for Sim when Sim asks for them. Sim writes output vectors via Envir, so one can redefine the output vector storing mechanism by changing Envir. Sim and its classes use Envir to print debug information.
- **Envir** ⇔ **Tkenv/Cmdenv**. Tkenv and Cmdenv are concrete user interface implementations. When a simulation program is started, the `main()` function (which is part of Envir) determines the appropriate user interface class, creates an instance and runs it by invoking its `run()` method. Sim's or the model's calls on the `ev` object are delegated to the user interface.

17.2 Embedding the OMNeT++ Simulation Kernel

This section discusses the issues of embedding the simulation kernel or a simulation model into a larger application. We assume that you do not just want to change one or two aspects

of the simulator (such as , event scheduling or result recording) or create a new user interface such as Cmdenv or Tkenv – if so, see chapter 16.

For the following section, we assume that you will write the embedding program from scratch. Meaning, starting from a `main()` function.

17.2.1 The main() Function

The minimalistic program described below initializes the simulation library and runs two simulations. In later sections we will review the details of the code and discuss how to improve it.

```
#include <omnetpp.h>

int main(int argc, char *argv[])
{
    // the following line MUST be at the top of main()
    cStaticFlag dummy;

    // initializations
    CodeFragments::executeAll(CodeFragments::STARTUP);
    SimTime::setScaleExp(-12);

    // load NED files
    cSimulation::loadNedSourceFolder("./foodir");
    cSimulation::loadNedSourceFolder("./bardir");
    cSimulation::doneLoadingNedFiles();

    // run two simulations
    simulate("FooNetwork", 1000);
    simulate("BarNetwork", 2000);

    // deallocate registration lists, loaded NED files, etc.
    CodeFragment::executeAll(CodeFragment::SHUTDOWN);
    return 0;
}
```

The first few lines of the code initialize the simulation library. The purpose of `cStaticFlag` is to set a global variable to `true` for the duration of the `main()` function, to help the simulation library handle exceptions correctly in extreme cases. `CodeFragment::executeAll(CodeFragment::STARTU` performs various startup tasks, such as building registration tables out of the `Define_Module()`, `Register_Class()` and similar entries throughout the code. `SimTime::setScaleExp(-12)` sets the simulation time resolution to picoseconds; other values can be used as well, but it is mandatory to choose one.

> **NOTE:** The simulation time exponent cannot be changed at a later stage, since it is a global variable, and the values of the existing `simtime_t` instances would change.

The code then loads the NED files from the `foodir` and `bardir` subdirectories of the working directory (as if the NED path was `./foodir;./bardir`), and runs two simulations.

17.2.2 The simulate() Function

A minimalistic version of the `simulate()` function is shown below. In order to shorten the code, the exception handling code has been ommited (`try`/`catch` blocks) apart from the event loop. However, every line is marked with "`E!`" where various problems with the simulation model can occur and can be thrown as exceptions.

```
void simulate(const char *networkName, simtime_t limit)
{
    // look up network type
    cModuleType *networkType = cModuleType::find(networkName);
    if (networkType == NULL) {
        printf("No such network: %s\n", networkName);
        return;
    }

    // create a simulation manager and an environment for the simulation
    cEnvir *env = new CustomSimulationEnv(argc, argv, new EmptyConfig());
    cSimulation *sim = new cSimulation("simulation", env);
    cSimulation::setActiveSimulation(sim);

    // set up network and prepare for running it
    sim->setupNetwork(networkType); //E!
    sim->startRun(); //E!

    // run the simulation
    bool ok = true;
    try {
        while (sim->getSimTime() < limit) {
            cSimpleModule *mod = sim->selectNextModule(); //E!
            if (!mod)
                break;
            sim->doOneEvent(mod);  //E!
        }
        printf("Finished: time limit reached\n");
    }
    catch (cTerminationException& e) {
        printf("Finished: %s\n", e.what());
    }
    catch (std::exception& e) {
        ok = false;
        printf("ERROR: %s\n", e.what());
    }

    if (ok)
        simulation.callFinish();  //E!

    // finish the simulation and clean up the network
    sim->endRun();  //E!
    sim->deleteNetwork();  //E!

    cSimulation::setActiveSimulation(NULL);
    delete sim; // deletes env as well
```

```
}
```

The function accepts a network type name (which must be fully qualified with a package name) and a simulation time limit.

In the first few lines the system looks up the network name among the modules that have been loaded from the NED files, and an error message is printed if it is not found.

Then it is required to create and activate a simulation manager object (`cSimulation`). The simulation manager requires another object, called the environment object. This environment object is used by the simulation manager to read the configuration. In addition, the results produced by the simulation manager are written to this environment object.

The environment object (`CustomSimulationEnv` in the above code) must be provided by the programmer; this is described in detail in a later section.

> **NOTE:** Before version 4.0, `simulation` and `ev` were global variables; In the current version they are macros that refer to `*cSimulation::getActiveSimulation()` and `*cSimulation::getActiveSimulation()->getEnvir()`.

The network is then set up in the simulation manager. The `sim->setupNetwork()` method creates the system module and recursively all modules and their interconnections; module parameters are also read from the configuration (where required) and assigned. If there is an error (for example, module type not found), an exception will be thrown. The exception object is some kind of `std::exception`, usually a `cRuntimeError`.

If the network setup was successful, the `sim->startRun()` function is called, and the `initialize()` methods of modules and channels are then activated. An exception is thrown if something goes wrong in any of the `initialize()` methods.

The following lines run the simulation by calling `sim->selectNextModule()` and `sim->doOneEvent()` in an event loop, until the simulation time limit is reached or an exception occurs. Exceptions that are subclassed from `cTerminationException` signify the normal termination of the simulation process; other exceptions indicate various errors.

If the simulation has completed successfully (`ok==true`), the code goes on to call the `finish()` methods of modules and channels. Then, regardless of whether there was an error, `sim->endRun()` is called, and the network is shut down using `sim->deleteNetwork()`.

Finally, the simulation manager object is deallocated, but the active simulation manager is not allowed to be deleted; therefore it is deactivated using `setActiveSimulation(NULL)`.

17.2.3 Providing an Environment Object

The environment object needs to be subclassed from the `cEnvir` class, but since it has many pure virtual methods, it is easier to begin by subclassing `cNullEnvir`. `cNullEnvir` defines all pure virtual methods with either an empty body or with a body that throws an "`unsupported method called`" exception. You can redefine methods to be more sophisticated later on, as you progress with the development.

You must redefine the `readParameter()` method. This enables module parameters to obtain their values. For debugging purposes, you can also redefine `sputn()` where module log messages are written to. `cNullEnvir` only provides one random number generator, so if your simulation model uses more than one, you also need to redefine the `getNumRNGs()` and `getRNG(k)` methods. To print or store simulation records, redefine `recordScalar()`, `recordStatistic()` and/or the output vector related methods. Other `cEnvir` methods are

invoked from the simulation kernel to inform the environment about messages being sent, events scheduled and cancelled, modules created, and so on.

The following example shows a minimalistic environment class that is enough to get started:

```
class CustomSimulationEnv : public cNullEnvir
{
  public:
    // constructor
    CustomSimulationEnv(int ac, char **av, cConfiguration *c) :
        cNullEnvir(ac, av, c) {}

    // model parameters: accept defaults
    virtual void readParameter(cPar *par) {
        if (par->containsValue())
            par->acceptDefault();
        else
            throw cRuntimeError("no value for %s", par->getFullPath().c_str());
    }

    // send module log messages to stdout
    virtual void sputn(const char *s, int n) {
        (void) ::fwrite(s,1,n,stdout);
    }
};
```

17.2.4 Providing a Configuration Object

The configuration object needs to subclass from `cConfiguration`. `cConfiguration` also has several methods, but the typed ones (`getAsBool()`, `getAsInt()`, etc.) have default implementations that delegate to the much fewer string-based methods (`getConfigValue()`, etc.).

It is fairly straightforward to implement a configuration class that emulates an empty ini file:

```
class EmptyConfig : public cConfiguration
{
  protected:
    class NullKeyValue : public KeyValue {
      public:
        virtual const char *getKey() const {return NULL;}
        virtual const char *getValue() const {return NULL;}
        virtual const char *getBaseDirectory() const {return NULL;}
    };
    NullKeyValue nullKeyValue;

  protected:
    virtual const char *substituteVariables(const char *value) {return value;}

  public:
    virtual const char *getConfigValue(const char *key) const
        {return NULL;}
    virtual const KeyValue& getConfigEntry(const char *key) const
        {return nullKeyValue;}
```

```
    virtual const char *getPerObjectConfigValue(const char *objectFullPath,
        const char *keySuffix) const {return NULL;}
    virtual const KeyValue& getPerObjectConfigEntry(const char *objectFullPath,
        const char *keySuffix) const {return nullKeyValue;}
};
```

17.2.5 Loading NED Files

NED files can be loaded with any of the following static methods of `cSimulation`: `loadNedSourceFolder()`, `loadNedFile()`, and `loadNedText()`. The first method loads an entire subdirectory tree, the second method loads a single NED file, and the third method takes a literal string containing NED code and parses it.

> **NOTE:** One use of `loadNedText()` is to parse NED sources previously converted to C++ string constants and linked into the executable. This enables creating executables that are self-contained, and do not require NED files to be distributed with them.

The above functions can also be mixed, but after the last call, `doneLoadingNedFiles()` must be invoked (it checks for unresolved NED types).

Loading NED files has a global effect; therefore they cannot be unloaded.

17.2.6 How to Eliminate NED Files

It is possible to get rid of NED files altogether. This would also remove the dependency on the `oppnedxml` library and the code in `sim/netbuilder`, although at the cost of additional coding.

> **NOTE:** When the only purpose is to get rid of NED files as external dependency of the program, it is simpler to use `loadNedText()` on NED files converted to C++ string constants instead.

The trick is to write `cModuleType` and `cChannelType` objects for your simple module, compound module and channel types, and register them manually. For example, `cModuleType` has pure virtual methods called `createModuleObject()`, `addParametersAndGatesTo(module)`, `setupGateVectors(module)`, `buildInside(module)`, which you need to implement. The body of the `buildInside()` method would be similar to C++ files generated by `nedtool` of OMNeT++ 3.x.

17.2.7 Assigning Module Parameters

As already mentioned, modules obtain values for their input parameters by calling the `readParameter()` method of the environment object (`cEnvir`).

> **NOTE:** `readParameter()` is only called for parameters that have not been set to a fixed (i.e. non-`default`) value in the NED files.

The `readParameter()` method should be written in a manner that enables it to assign the parameter. When doing so, it can recognize the parameter from its name (`par->getName()`),

from its full path (`par->getFullPath()`), from the owner module's class (`par->getOwner()->getClassName()`) or NED type name (`((cComponent *)par->getOwner())->getNedTypeName()`). Then it can set the parameter using one of the typed setter methods (`setBoolValue()`, `setLongValue()`, etc.), or set it to an expression provided in string form (`parse()` method). It can also accept the default value if it exists (`acceptDefault()`).

The following code is a straightforward example that answers parameter value requests from a pre-filled table.

```
class CustomSimulationEnv : public cNullEnvir
{
  protected:
    // parameter (fullpath,value) pairs, needs to be pre-filled
    std::map<std::string,std::string> paramValues;
  public:
    ...
    virtual void readParameter(cPar *par) {
        if (paramValues.find(par->getFullPath())!=paramValues.end())
            par->parse(paramValues[par->getFullPath()]);
        else if (par->containsValue())
            par->acceptDefault();
        else
            throw cRuntimeError("no value for %s", par->getFullPath().c_str());
    }
};
```

17.2.8 Extracting Statistics from the Model

There are several ways you can extract statistics from the simulation.

C++ Calls into the Model

Modules in the simulation are C++ objects. If you add the appropriate public getter methods to the module classes, you can call them from your main program to obtain statistics. Modules may be looked up with the `getModuleByPath()` method of `cSimulation`, then cast to the specific module type via `check_and_cast<>()` so that the getter methods can be invoked.

```
cModule *mod = simulation.getModuleByPath("Network.client[2].app");
WebApp *appMod = check_and_cast<WebApp *>(mod);
int numRequestsSent = appMod->getNumRequestsSent();
double avgReplyTime = appMod->getAvgReplyTime();
...
```

The drawback of this approach is that getters need to be added manually to all affected module classes, which might not be practical, especially if modules come from external projects.

`cEnvir` Callbacks

A more general way is to catch `recordScalar()` method calls in the simulation model. The `cModule`'s `recordScalar()` method delegates to the similar function in `cEnvir`. You may define the latter function so that it stores all recorded scalars (for example in an `std::map`),

where the main program can find them later. Values from output vectors can be captured in a similar manner.

An example implementation:

```
class CustomSimulationEnv : public cNullEnvir
{
  private:
    std::map<std::string, double> results;
  public:
    virtual void recordScalar(cComponent *component, const char *name,
                              double value, opp_string_map *attributes=NULL)
    {
       results[component->getFullPath()+"."+name] = value;
    }

    const std::map<std::string, double>& getResults() {return results;}
};

...

const std::map<std::string, double>& results = env->getResults();
int numRequestsSent = results["Network.client[2].app.numRequestsSent"];
double avgReplyTime = results["Network.client[2].app.avgReplyTime"];
```

A drawback of this approach is that compile-time checking of statistics names is lost, but the advantages are that any simulation model can now be used without changes, and that capturing additional statistics does not require code modification in the main program.

17.2.9 The Simulation Loop

To run the simulation, the `selectNextModule()` and `doOneEvent` methods of `cSimulation` must be called in a loop:

```
while (sim->getSimTime() < limit)
{
    cSimpleModule *mod = sim->selectNextModule();
    sim->doOneEvent(mod);
}
```

Depending on the concrete scheduler class, the `selectNextModule()` may return `NULL`. The default `cSequentialScheduler` never returns `NULL`.

The execution may terminate in various ways. Runtime errors cause a `cRuntimeError` (or other kind of `std::exception`) to be thrown. `cTerminationException` is thrown on normal termination conditions, such as when the simulation runs out of events to process.

You may customize the loop to exit on other termination conditions as well, such as on a simulation time limit (see above), on a CPU time limit, or when results reach a required accuracy. It is relatively straightforward to build in progress reporting and interactivity (start/stop).

Animation can be hooked up to the appropriate callback methods of `cEnvir`: `beginSend()`, `sendHop()`, `endSend()`, and others.

17.2.10 Multiple, Coexisting Simulations

It is possible for several instances of `cSimulation` to coexist, and also to set up and simulate a network in each instance. However, this requires frequent use of `cSimulation::setActiveSimulation()`. Before invoking any `cSimulation` method or module method, the corresponding `cSimulation` instance needs to be designated as the active simulation manager. This is necessary because several models and simulation kernel methods refer to the active simulation manager instance via the `simulation` macro, and it is similar with the `ev` macro.

> **NOTE:** Before the 4.0 version, `simulation` and `ev` were global variables; in the current version they are macros that refer to `*cSimulation::getActiveSimulation()` and `*cSimulation::getActiveSimulation()->getEnvir()`.

Every `cSimulation` instance should have its own associated environment object (`cEnvir`). Environment objects may not be shared among several `cSimulation` instances. The `cSimulation`'s destructor also removes the associated `cEnvir` instance.

`cSimulation` instances may be reused from one simulation to another, but it is also possible to create a new instance for each simulation run.

> **NOTE:** It is not possible to run different simulations concurrently from different theads, due to the use of global variables which are not easy to eliminate, such as the active simulation manager pointer and the active environment object pointer. Static buffers and objects (like string pools) are also used for efficiency reasons in some places inside the simulation kernel.

17.2.11 Installing a Custom Scheduler

The default event scheduler is `cSequentialScheduler`. To replace it with a different scheduler (e.g. `cRealTimeScheduler` or your own scheduler class), add a `setScheduler()` call into `main()`:

```
cScheduler *scheduler = new CustomScheduler();
simulation.setScheduler(scheduler);
```

It is usually not a good idea to change schedulers in the middle of a simulation, therefore `setScheduler()` may only be called when no network is set up.

17.2.12 Multi-Threaded Programs

The OMNeT++ simulation kernel is not reentrant; therefore it must be protected against concurrent access.

Appendix A

NED Reference

A.1 Syntax

A.1.1 NED File Name Extension

NED files have the `.ned` file name suffix. This is mandatory, and cannot be overridden.

A.1.2 NED File Encoding

NED files are ASCII, but non-ASCII characters are permitted in comments and string literals. This allows for using encodings that are a superset of ASCII, for example ISO 8859-1 and UTF-8.

> **NOTE:** There is no standard way to specify or determine the encoding of a NED file. It is up to the user to configure the desired encoding in text editors and other tools that edit or process NED files.

String literals (e.g. in parameter values) will be passed to the C++ code as `const char *` without any conversion; it is up to the simulation model to interpret them using the desired encoding.

Line ending may be either CR or CRLF, regardless of the platform.

A.1.3 Reserved Words

NED file authors have to take care that no reserved words are used as identifiers. The reserved words of the NED language are:

```
allowunconnected bool channel channelinterface connections const default dou-
ble extends false for gates if import index inout input int like module mod-
uleinterface network output package parameters property simple sizeof string
submodules this true typename types volatile xml xmldoc
```

A.1.4 Identifiers

Identifiers must be composed of letters of the English alphabet (a-z, A-Z), numbers (0-9) and underscore "_". Identifiers may only begin with a letter or underscore.

The recommended way to compose identifiers from multiple words is to capitalize the beginning of each word (*camel case*).

A.1.5 Case Sensitivity

Keywords and identifiers in the NED language are case sensitive. For example, `TCP` and `Tcp` are two different names.

A.1.6 Literals

String Literals

String literals use double quotes. The following C-style backslash escapes are recognized: `\b`, `\f`, `\n`, `\r`, `\t`, `\\`, `\"`, and `\xhh` where *h* is a hexadecimal digit.

Numeric Constants

Numeric constants are accepted in the usual decimal, hexadecimal (`0x` prefix) and scientific notations. Octal numbers are not accepted (numbers that start with the `0` digit are interpreted as decimal.)

Quantity Constants

A quantity constant has the form *(<numeric-constant> <unit>)+*, for example `12.5mW` or `3h 15min 37.2s`. Whitespace is optional in front of a unit, but must be present after a unit if it is followed by a number.

When multiple measurement units are present, they have to be convertible into each other (i.e. refer to the same physical quantity).

Section A.5.6 lists the units recognized by OMNeT++. Other units can be used as well; the only downside being that OMNeT++ will not be able to perform conversions on them.

A.1.7 Comments

Comments can be placed at the end of lines. Comments begin with a double slash `//`, and continue until the end of the line.

A.1.8 Grammar

The grammar of the NED language can be found in Appendix B.

A.2 Built-in Definitions

The NED language has the following built-in definitions, all in the `ned` package: channels `IdealChannel`, `DelayChannel`, and `DatarateChannel`; module interfaces `IBidirectionalChannel`, and `IUnidirectionalChannel`. The latter two are reserved for future use.

The bodies of `@statistic` properties have been omitted for brevity from the following listing.

NOTE: You can print the full definitions by running `opp_run -h neddecls`.

```
package ned;
@namespace("");

channel IdealChannel
{
    @class(cIdealChannel);
}

channel DelayChannel
{
    @class(cDelayChannel);
    @signal[messageSent](type=cMessage);
    @signal[messageDiscarded](type=cMessage);
    @statistic[messages](...);
    @statistic[messagesDiscarded](...);
    bool disabled = default(false);
    double delay = default(0s) @unit(s); // propagation delay
}

channel DatarateChannel
{
    @class(cDatarateChannel);
    @signal[channelBusy](type=long);
    @signal[messageSent](type=cMessage);
    @signal[messageDiscarded](type=cMessage);
    @statistic[busy](...);
    @statistic[utilization](...);
    @statistic[packets](...);
    @statistic[packetBytes](...);
    @statistic[packetsDiscarded](...);
    @statistic[throughput](...);
    bool disabled = default(false);
    double delay = default(0s) @unit(s); // propagation delay
    double datarate = default(0bps) @unit(bps); // bits per second; 0=infinite
    double ber = default(0); // bit error rate (BER)
    double per = default(0); // packet error rate (PER)
}

moduleinterface IBidirectionalChannel
{
    gates:
        inout a;
```

```
        inout b;
}

moduleinterface IUnidirectionalChannel
{
    gates:
        input i;
        output o;
}
```

A.3 Packages

NED supports hierarchical namespaces called *packages*. The model is similar to Java packages, with minor changes.

A.3.1 Package Declaration

A NED file may contain a package declaration. The package declaration uses the **package** keyword, and specifies the package for the definitions in the NED file. If there is no package declaration, the file's contents are in the *default package*.

Component type names must be unique within their package.

A.3.2 Directory Structure, package.ned

Like in Java, the directory of a NED file must match the package declaration. However, it is possible to omit directories at the top which do not contain any NED files (like the typical `/org/<projectname>` directories in Java).

The top of a directory tree containing NED files is named a *NED source folder*.

> **NOTE:** The OMNeT++ runtime recognizes a `NEDPATH` environment variable, which contains a list of NED source folders, and is similar to the Java `CLASSPATH` variable. `NEDPATH` also has a command-line option equivalent.

The `package.ned` file at the top level of a NED source folder plays a special role.

If there is no toplevel `package.ned` or it contains no package declaration, the declared package of a NED file in the folder `<srcfolder>/x/y/z` *must* be `x.y.z`. If there is a toplevel `package.ned` and it declares the package as `a.b`, then any NED file in the folder `<srcfolder>/x/y/z` *must* have the declared package `a.b.x.y.z`.

> **NOTE:** `package.ned` files are allowed in other folders as well. They may contain properties and/or documentation for their package, but cannot be used to define the package they are in.

A.4 Components

Simple modules, compound modules, networks, channels, module interfaces and channel interfaces are called *components*.

A.4.1 Simple Modules

Simple module types are declared with the **`simple`** keyword; see the NED Grammar (Appendix B) for the syntax.

Simple modules may have properties (A.4.8), parameters (A.4.9) and gates (A.4.11).

A simple module type may not have inner types (A.4.15).

A simple module type may extend another simple module type, and may implement one or more module interfaces (A.4.5). Inheritance rules are described in section A.4.21, and interface implementation rules in section A.4.20.

Every simple module type has an associated C++ class, which must be subclassed from `cSimpleModule`. The way of associating the NED type with the C++ class is described in section A.4.7.

A.4.2 Compound Modules

Compound module types are declared with the **`module`** keyword; see the NED Grammar (Appendix B) for the syntax.

A compound module may have properties (A.4.8), parameters (A.4.9), and gates (A.4.11); its internal structure is defined by its submodules (A.4.12) and connections (A.4.13); and it may also have inner types (A.4.15) that can be used for its submodules and connections.

A compound module type may extend another compound module type, and may implement one or more module interfaces (A.4.5). Inheritance rules are described in section A.4.21, and interface implementation rules in section A.4.20.

A.4.3 Networks

The network Keyword

A network declared with the **`network`** keyword is equivalent to a compound module (**`module`** keyword) with the `@isNetwork(true)` property.

> **NOTE:** A simple module can only be designated to be a network by spelling out the `@isNetwork` property; the **`network`** keyword cannot be used for that purpose.

The @isNetwork Property

The `@isNetwork` property is only recognized for simple modules and compound modules. The value may be empty, true or false:

```
@isNetwork;
@isNetwork();
@isNetwork(true);
```

```
@isNetwork(false);
```

The empty value corresponds to `@isNetwork(true)`.

The `@isNetwork` property is not inherited; that is, a subclass of a module with `@isNetwork` set does not automatically become a network. The `@isNetwork` property needs to be explicitly added to the subclass to make it a network.

> **Rationale:** Subclassing may introduce changes to a module that make it unfit to be used as a network.

A.4.4 Channels

Channel types are declared with the **`channel`** keyword; see the NED Grammar (Appendix B) for the syntax.

Channel types may have properties (A.4.8) and parameters (A.4.9).

A channel type may not have inner types (A.4.15).

A channel type may extend another channel type, and may implement one or more channel interfaces (A.4.6). Inheritance rules are described in section A.4.21, and interface implementation rules in section A.4.20.

Every channel type has an associated C++ class, which must be subclassed from `cChannel`. The way of associating the NED type with the C++ class is described in section A.4.7.

The `@defaultname` property of a channel type determines the default name of the channel object when used in a connection.

A.4.5 Module Interfaces

Module interface types are declared with the **`moduleinterface`** keyword; see the NED Grammar (Appendix B) for the syntax.

Module interfaces may have properties (A.4.8), parameters (A.4.9), and gates (A.4.11). However, parameters are not allowed to have a value assigned, not even a default value.

A module interface type may not have inner types (A.4.15).

A module interface type may extend one or more other module interface types. Inheritance rules are described in section A.4.21.

A.4.6 Channel Interfaces

Channel interface types are declared with the **`channelinterface`** keyword; see the NED Grammar (Appendix B) for the syntax.

Channel interfaces may have properties (A.4.8) and parameters (A.4.9). However, parameters are not allowed to have a value assigned, not even a default value.

A channel interface type may not have inner types (A.4.15).

A channel interface type may extend one or more other channel interface types. Inheritance rules are described in section A.4.21.

A.4.7 Resolving the C++ Implementation Class

The procedure for determining the C++ implementation class for simple modules and for channels are identical. It goes as follows (we are going to say *component* instead of *"simple module or channel"*):

If the component extends another component and has no `@class` property, the C++ implementation class is inherited from the base type.

If the component contains a `@class` property, the C++ class name will be composed of the *current namespace* (see below) and the value of the `@class` property. The `@class` property should contain a single value.

> **NOTE:** The `@class` property may itself contain a namespace declaration (ie. may contain "`::`").

If the component contains no `@class` property and has no base class, the C++ class name will be composed of the *current namespace* and the unqualified name of the component.

> **IMPORTANT:** NED subclassing does not imply subclassing the C++ implementation! If you want to subclass a simple module or channel in NED as well as in C++, you explicitly need to specify the `@class` property, otherwise the derived simple module or channel will continue to use the C++ class from its super type.

Compound modules will be instantiated with the built-in `cModule` class, unless the module contains the `@class` property. When `@class` is present, the resolution rules are the same as with simple modules.

Current Namespace

The *current namespace* is the value of the first `@namespace` property found while searching the following order:

1. the current NED file
2. the `package.ned` file in the current package or the first ancestor package searching upwards

> **NOTE:** Note that namespaces coming from multiple `@namespace` properties in different scopes do not nest, but rather, the nearest one wins.

The `@namespace` property should contain a single value.

A.4.8 Properties

Properties are a means of adding metadata annotations to NED files, component types, parameters, gates, submodules, and connections.

Identifying a Property

Properties are identified by name. It is possible to have several properties on the same object with the same name, as long as they have unique indices. An index is an identifier in square brackets after the property name.

The following example shows a property without index, one with the index `index1`, and a third with the index `index2`.

```
@prop1();
@prop2[index1]();
@prop3[index2]();
```

Property Value

The value of the property is specified inside parentheses. The property value consists of *key=valuelist* pairs, separated by semicolons; *valuelist* elements are separated with commas. Example:

```
@prop(key1=value11,value12,value13;key2=value21,value22)
```

Keys must be unique.

If the key+equal sign part (`key=`) is missing, the valuelist belongs to the *default key*. Examples:

```
@prop1(value1,value2)
@prop2(value1,value2;key1=value11,value12,value13)
```

Most of the properties use the default key with one value. Examples:

```
@namespace(inet);
@class(Foo);
@unit(s);
```

Property values have a liberal syntax (see Appendix B). Values that do not fit the grammar (notably, those containing a comma or a semicolon) need to be surrounded with double quotes.

When interpreting a property value, one layer of quotes is removed automatically, that is, `foo` and `"foo"` are the same. Within quotes, escaping works in the same way as within string literals (see A.1.6).

Example:

```
@prop(marks=the ! mark, "the , mark", "the ; mark", other marks); // 4 items
```

Placement

Properties may be added to NED files, component types, parameters, gates, submodules and connections. For the exact syntax, see Appendix B.

When a component type extends another component type(s), properties are merged. This is described in section A.4.21.

Property Declarations

The **property** keyword is reserved for future use. It is envisioned that accepted property names and property keys would need to be pre-declared, so that the NED infrastructure can warn the user about mistyped or unrecognized names.

A.4.9 Parameters

Parameters can be defined and assigned in the **parameters** section of component types. In addition, parameters can also be assigned in the **parameters** sections of submodule bodies and connection bodies, but those places do not allow adding new parameters.

The **parameters** keyword is optional, and can be omitted without change in the meaning.

The **parameters** section may also hold pattern assignments (A.4.10) and properties (A.4.8).

A parameter is identified by a name, and has a data type. A parameter may have value or default value, and may also have properties (see A.4.8).

Accepted parameter data types are **double**, **int**, **string**, **bool**, and **xml**. Any of the above types can be declared **volatile** as well (`volatile int`, `volatile string`, etc.)

The presence of a data type keyword determines whether the given line defines a new parameter or refers to an existing parameter. One can assign a value or default value to an existing parameter, and/or modify its properties or add new properties.

Examples:

```
int a;            // defines new parameter
int b @foo;       // new parameter with property
int c = default(5); // new parameter with default value
int d = 5;        // new parameter with value assigned
int e @foo = 5;   // new parameter with property and value
f = 10;           // assignment to existing (e.g.inherited) parameter
g = default(10); // overrides default value of existing parameter
h;                // legal, but does nothing
i @foo(1);        // adds a property to existing parameter
j @foo(1) = 10;   // adds a property and value to existing parameter
```

Parameter values are NED expressions. Expressions are described in section A.5.

For **volatile** parameters, the value expression is evaluated every time the parameter value is accessed. Non-**volatile** parameters are evaluated only once.

> **NOTE:** The **const** keyword is reserved for future use within expressions to define constant subexpressions, i.e. to denote a part within an expression that should only be evaluated once. Constant subexpressions are not supported yet.

The following properties are recognized for parameters: `@unit`, `@prompt`.

The @prompt Property

The `@prompt` property defines a prompt string for the parameter. The prompt string is used when/if a simulation runtime user interface interactively prompts the user for the parameter's value.

The `@prompt` property is expected to contain one string value for the default key.

The @unit Property

A parameter may have a `@unit` property to associate it with a measurement unit. The `@unit` property should contain one string value for the default key. Examples:

```
@unit(s)
@unit(second)
```

When present, values assigned to the parameter must be in the same or in a compatible (that is, convertible) unit. Examples:

```
double a @unit(s) = 5s;   // OK
double a @unit(s) = 10ms; // OK; will be converted to seconds
double a @unit(s) = 5;    // error: should be 5s
double a @unit(s) = 5kg;  // error: incompatible unit
```

`@unit` behavior for non-numeric parameters (boolean, string, XML) is unspecified (may be ignored or may be an error).

The `@unit` property of a parameter may not be modified via inheritance.

Example:

```
simple A {
    double p @unit(s);
}
simple B extends A {
    p @unit(mW);  // illegal: cannot override @unit
}
```

A.4.10 Pattern Assignments

Pattern assignments allow one to set more than one parameter using wildcards, and to assign parameters deeper down in a submodule tree. Pattern assignments may occur in the **`parameters`** section of component types, submodules and connections.

The syntax of a pattern assignment is <*pattern*> = <*value*>.

A pattern consists of two or more pattern elements, separated by dots. The pattern element syntax is defined so that it can accomodate names of parameters, submodules (optionally with index), gates (optionally with the `$i`/`$o` suffix and/or index) and connections, and their wildcard forms. (The default name of connection channel objects is **`channel`**.)

Wildcard forms may use:

1. Asterisks: They match zero or more characters except dots.
2. Numeric ranges, `{`<*start*>`..`<*end*>`}` e.g. `{5..120}` or `{..10}`. They match numbers embedded in identifiers, that is, a sequence of decimal digit characters interpreted as a nonnegative integer that is within the specified *start..end* range (both limits are inclusive). Both *start* and *end* are optional.
3. Numeric index ranges, `[`<*start*>`..`<*end*>`]`. e.g. `[5..120]` or `[..10]`. They are intended for selecting submodule and gate index ranges. They match a nonnegative integer enclosed in square brackets that is within the specified *start..end* range (both limits are inclusive). Both *start* and *end* are optional.

4. Double asterisks: They match zero or more characters (including dots), and can be used to match more than one parameter path elements.

See the NED language grammar (Appendix B) for a more formal definition of the pattern syntax.

Examples:

```
host1.tcp.mss = 512B;
host*.tcp.mss = 512B; // matches host, host1, host2, hostileHost, ...
host{9..11}.tcp.mss = 512B; // matches host9/host10/host11, but nothing else
host[9..11].tcp.mss = 512B; // matches host[9]/host[10]/host[11], but nothing else
**.mss = 512B; // matches foo.mss, host[1].transport.tcp[0].mss, ...
```

A.4.11 Gates

Gates can be defined in the **gates** section of component types. The size of a gate vector (see below) may be specified at the place of defining the gate, via inheritance in a derived type, and also in the **gates** block of a submodule body. A submodule body does not allow defining new gates.

A gate is identified by a name, and is characterized by a type (**input**, **output**, **inout**) and optionally a vector size. Gates may also have properties (see A.4.8).

Gates may be scalar or vector. The vector size is specified with a numeric expression inside square brackets. The vector size may also be left unspecified by writing an empty pair of square brackets.

An already specified gate vector size may not be overridden in subclasses or in a submodule.

The presence of a gate type keyword determines whether the given line defines a new gate or refers to an existing gate. One can specify the gate vector size for an existing gate vector, and/or modify its properties, or add new properties.

Examples:

```
gates:
  input a;          // defines new gate
  input b @foo;     // new gate with property
  input c[];        // new gate vector with unspecified size
  input d[8];       // new gate vector with size=8
  e[10];            // set gate size for existing (e.g.inherited) gate vector
  f @foo(bar);      // add property to existing gate
  g[10] @foo(bar); // set gate size and add property to existing gate
```

Gate vector sizes are NED expressions. Expressions are described in section A.5.

See the Connections section (A.4.13) for more information on gates.

Recognized Gate Properties

The following properties are recognized for gates: `@directIn` and `@loose`. They have the same effect: When either of them is present on a gate, the gate is not required to be connected in the connections section of a compound module (see A.4.13).

`@directIn` should be used when the gate is an **`input`** gate that is intended for being used as a target for the `sendDirect()` method; `@loose` should be used in any other case when the gate is not required to be connected for some reason.

> **NOTE:** The reason `@directIn` gates are not *required* to remain unconnected is that it is often useful to wrap such modules in a compound module, where the compound module also has a `@directIn` input gate that is internally connected to the submodule's corresponding gate.

Example:

```
gates:
    input radioIn @directIn;
```

A.4.12 Submodules

Submodules are defined in the **`submodules`** section of the compound module.

The type of the submodule may be specified statically or parametrically.

Submodules may be scalar or vector. The size of submodule vectors must be specified as a numeric expression inside square brackets.

Submodules may also be conditional.

A submodule definition may or may not have a body (a curly brace delimited block). An empty submodule body is equivalent to a missing one.

Syntax examples:

```
submodules:
    ip : IP;          // scalar submodule without body
    tcp : TCP {}      // scalar submodule with empty body
    app[10] : App;    // submodule vector
```

Submodule Type

The simple or compound module type (A.4.1, A.4.2) that will be instantiated as the submodule may be specified either statically (with a concrete module type name) or parametrically.

Static Submodule Type

Submodules with a statically defined type are those that contain a concrete NED module type name. Example:

```
tcp : TCP;
```

See section A.4.18 for the type resolution rules.

Parametric Submodule Type

Parametric submodule type means that the NED type name is given in a string expression. The string expression may be specified locally in the submodule declaration, or elsewhere using typename patterns (see later).

Parametric submodule types are syntactically denoted by the presence of a pair of angle brackets and the **like** keyword. The angle brackets contain the string expression; or if the type is not specified locally, they may be empty or may contain a default value for the type name (`default(...)` syntax). The syntax mandates that you also specify a module interface type A.4.5 (after the **like** keyword), which the concrete module type must implement in order for it to be eligible to be chosen.

Examples:

```
tcp : <tcpType> like ITCP;         // type comes from parent module parameter
tcp : <"TCP_"+suffix> like ITCP;   // expression using parent module parameter

tcp : <> like ITCP;                // type must be specified elsewhere

tcp : <default("TCP")> like ITCP; // type may be specified elsewhere;
                                   // if not, the default is "TCP"
tcp : <default("TCP_"+suffix)> like ITCP;
                                   // type may be specified elsewhere;
                                   // if not, the default is an expression
```

See the NED Grammar (Appendix B) for the formal syntax, and section A.4.19 for the type resolution rules.

Conditional Submodules

Submodules may be made conditional using the **if** keyword. The condition expression must evaluate to a boolean; if the result is `false`, the submodule is not created, and trying to connect its gates or reference its parameters will be an error.

An example:

```
submodules:
  tcp : TCP if withTCP { ... }
```

Parameters, Gates

A submodule body may contain parameters (A.4.9) and gates (A.4.5).

A submodule body cannot define new parameters or gates. It is only allowed to assign existing parameters, and to set the vector size of existing gate vectors.

It is also allowed to add or modify submodule properties and parameter/gate properties.

A.4.13 Connections

Connections are defined in the **connections** section of the compound module.

Connections may not span multiple hierarchy levels, that is, a connection may be created between two submodules, a submodule and the compound module, or between two gates of the compound module.

Normally, all gates must be connected, including submodule gates and the gates of the compound module. When the **allowunconnected** modifier is present after **connections**, gates will be allowed to be left unconnected.

NOTE: The `@directIn` and `@loose` gate properties are alternatives to the `connections allowunconnected` syntax; see A.4.11.

Connections may be conditional, and may be created using loops (see A.4.14).

Connection Syntax

The connection syntax uses arrows (`-->`, `<--`) to connect **`input`** and **`output`** gates, and double arrows (`<-->`) to connect **`inout`** gates. The latter is also said to be a bidirectional connection.

Arrows point from the source gate (a submodule output gate or a compound module input gate) to the destination gate (a submodule input gate or a compound module output gate). Connections may be written either left to right or right to left, that is, `a-->b` is equivalent to `b<--a`.

Gates are specified as *<modulespec>.<gatespec>* (to connect a submodule), or as *<gatespec>* (to connect the compound module). *<modulespec>* is either a submodule name (for scalar submodules), or a submodule name plus an index in square brackets (for submodule vectors). For scalar gates, *<gatespec>* is the gate name; for gate vectors it is either the gate name plus a numeric index expression in square brackets, or *<gatename>*`++`.

The *<gatename>*`++` notation causes the first unconnected gate index to be used. If all gates of the given gate vector are connected, the behavior is different for submodules and for the enclosing compound module. For submodules, the gate vector expands by one. For the compound module, it is an error to use `++` on a gate vector with no unconnected gates.

Syntax examples:

```
connections:
   a.out --> b.in;   // unidirectional between two submodules
   c.in[2] <-- in;   // parent-to-child; gate vector with index
   d.g++ <--> e.g++; // bidirectional, auto-expanding gate vectors
```

Rationale: The reason it is not supported to expand the gate vector of the compound module is that the module structure is built in top-down order: new gates would be left unconnected on the outside, as there is no way in NED to "go back" and connect them afterwards.

When the `++` operator is used with `$i` or `$o` (e.g. `g$i++` or `g$o++`, see later), it will actually add a gate pair (input+output) to maintain equal gate size for the two directions.

The syntax to associate a channel (see A.4.4) with the connection is to use two arrows with a channel specification in between (see later). The same syntax is used to add properties such as `@display` to the connection.

Inout Gates

An inout gate is represented as a gate pair: an input gate and an output gate. The two sub-gates may also be referenced and connected individually, by adding the `$i` and `$o` suffix to the name of the inout gate.

A bidirectional connection (which uses a double arrow to connect two inout gates), is also a shorthand for two uni-directional connections; that is,

```
a.g <--> b.g;
```

is equivalent to

```
a.g$o --> b.g$i;
a.g$i <-- b.g$o;
```

In inout gate vectors, gates are always in pairs, that is, `sizeof(g$i)==sizeof(g$o)` always holds. It is maintained even when `g$i++` or `g$o++` is used: the `++` operator will add a gate pair, not just an input or an output gate.

Specifying Channels

A channel specification associates a channel object with the connection. A channel object is an instance of a channel type (see A.4.4).

> **NOTE:** As bidirectional connections are a shorthand for a pair of uni-directional connections, they will actually create *two* channel objects, one for each direction.

The channel type to be instantiated may be implicit, or may be specified statically or parametrically.

A connection may have a body (a curly brace delimited block) for setting properties and/or parameters of the channel.

A connection syntax allows one to specify a name for the channel object. When not specified, the channel name will be taken from the `@defaultname` property of the channel type; when there is no such property, it will be `"channel"`. Custom connection names can be useful for easier addressing of channel objects when assigning parameters using patterns.

See subsequent sections for details.

Implicit Channel Type

If the connection syntax does not say anything about the channel type, it is implicitly determined from the set of connection parameters used.

Syntax examples for connections with implicit channel types:

```
a.g <--> b.g; // no parameters
a.g <--> {delay = 1ms;} <--> b.g;  // assigns delay
a.g <--> {datarate = 100Mbps; delay = 50ns;} <--> b.g; // assigns delay and datarate
```

For such connections, the actual NED type to be used will depend on the parameters set in the connection:

1. When no parameters are set, `ned.IdealChannel` is chosen.
2. When only `ned.DelayChannel` parameters are used (`delay` and `disabled`), `ned.DelayChannel` is chosen.
3. When only `ned.DatarateChannel` parameters are used (`datarate`, `delay`, `ber`, `per`, `disabled`), the chosen channel type will be `ned.DatarateChannel`.

Connections with implicit channel types may not use any other parameter.

Static Channel Type

Connections with a statically defined channel type are those that contain a concrete NED channel type name.

Examples:

```
a.g <--> FastEthernet <--> b.g;
a.g <--> FastEthernet {per = 1e-6;} <--> b.g;
```

See section A.4.18 for the type resolution rules.

Parametric Channel Type

Parametric channel types are similar to parametric submodule types, described in section A.4.12.

Parametric channel type means that the NED type name is given in a string expression. The string expression may be specified locally in the connection declaration, or elsewhere using typename patterns (see later).

Parametric channel types are syntactically denoted by the presence of a pair of angle brackets and the **like** keyword. The angle brackets contain the string expression; or if the type is not specified locally, they may be empty or may contain a default value for the type name (`default(...)` syntax). The syntax mandates that you also specify a channel interface type A.4.5 (after the **like** keyword), which the concrete channel type must implement in order for it to be eligible to be chosen.

Examples:

```
a.g++ <--> <channelType> like IMyChannel <--> b.g++;
                                    // type comes from parent module parameter
a.g++ <--> <"Ch_"+suffix> like IMyChannel <--> b.g++;
                                    // expression using parent module parameter
a.g++ <--> <> like IMyChannel <--> b.g++;
                                    // type must be specified elsewhere
a.g++ <--> <default("MyChannel")> like IMyChannel <--> b.g++;
                                    // type may be specified elsewhere;
                                    // if not, the default is "MyChannel"
a.g++ <--> <default("Ch_"+suffix)> like IMyChannel <--> b.g++;
                                    // type may be specified elsewhere;
                                    // if not, the default is an expression
```

See the NED Grammar (Appendix B) for the formal syntax, and section A.4.19 for the type resolution rules.

Channel Parameters and Properties

A channel definition may or may not have a body (a curly brace delimited block). An empty channel body (`{ }`) is equivalent to a missing one.

A channel body may contain parameters (A.4.9).

A channel body cannot define new parameters. It is only allowed to assign existing parameters.

It is also allowed to add or modify properties and parameter properties.

A.4.14 Conditional and Loop Connections, Connection Groups

The connections section may contain any number of connections and connection groups. A connection group is one or more connections grouped with curly braces.

Both connections and connection groups may be conditional (**if** keyword) or may be multiple (**for** keyword).

Any number of **for** and **if** clauses may be added to a connection or connection loop; they are interpreted as if they were nested in the given order. Loop variables of a **for** may be referenced from subsequent conditions and loops as well as in module and gate index expressions in the connections.

See the NED Grammar (B) for the exact syntax.

Example connections:

```
a.out --> b.in;
c.out --> d.in if p>0;
e.out[i] --> f[i].in for i=0..sizeof(f)-1, if i%2==0;
```

Example connection groups:

```
if p>0 {
    a.out --> b.in;
    a.in <-- b.out;
}
for i=0..sizeof(c)-1, if i%2==0 {
    c[i].out --> out[i];
    c[i].in <-- in[i];
}
for i=0..sizeof(d)-1, for j=0..sizeof(d)-1, if i!=j {
    d[i].out[j] --> d[j].in[i];
}
for i=0..sizeof(e)-1, for j=0..sizeof(e)-1 {
    e[i].out[j] --> e[j].in[i] if i!=j;
}
```

A.4.15 Inner Types

Inner types can be defined in the **types** section of compound modules, with the same syntax as toplevel (i.e. non-inner) types.

Inner types may not contain further inner types, that is, type nesting is limited to two levels.

Inner types are only visible inside the enclosing component type and its subclasses.

A.4.16 Name Uniqueness

Identifier names within a component must be unique. That is, the following items in a component are considered to be in the same name space and must not have colliding names:

- parameters
- gates

- submodules
- inner types
- the above items of super type(s)

For example, a gate and a submodule cannot have the same name.

A.4.17 Parameter Assignment Order

A module or channel parameter may be assigned in **parameters** blocks (see A.4.9) at various places in NED: in the module or channel type that defines it; in the type's subclasses; in the submodule or connection that instantiates the type. The parameter may also be assigned using pattern assignments (see A.4.10) in any compound module that uses the given module or channel type directly or indirectly.

Patterns are matched against the relative path of the parameter, which is the relative path of its submodule or connection, with a dot and the parameter name appended. The relative path is composed of a list of submodule names (name plus index) separated by dots; a connection is identified by the full name of its source gate plus the name of the channel object (which is currently always `channel`) separated by a dot.

> **NOTE:** As bidirectional connections are a shorthand for two unidirectional connections, the source gate name is qualified with `$i` or `$o` in the relative path.

Note that the **parameters** keyword itself is optional, and is usually not written out in submodules and connections.

This section describes the module and channel parameter assignments procedure.

The general rules are the following:

1. A (non-default) parameter assignment may not be overridden later; that is, if there are assignments in multiple places, the assignment "closest" to the parameter declaration will be effective; others will be flagged as errors.
2. A default value is only used if a non-default value is not present for the given parameter. A non-default value may also come from a source external to NED, namely the simulation configuration (`omnetpp.ini`).
3. Unlike non-default values, a default value *may* be overridden; that is, if there are default value assignments in multiple places, the assignment "farthest" from the parameter declaration will win.
4. Among pattern assignments within the same **parameters** block, the first match will win. Pattern assignments with default and non-default values are considered to be two disjoint sets, only one of which are searched at a time.

This yields the following conceptual search order for non-default parameter assignments:

1. First, the NED type that contains the parameter declaration is checked;
2. Then its subclasses are checked;
3. Then the submodule or connection that instantiates the type is checked;

4. Then the compound module that contains the submodule or connection is checked for matching pattern assignments;
5. Then, assuming the compound module is part of a network, the search for matching pattern assignments continues up on the module tree until the root (the module that represents the network). At each level (compound module), first the specific submodule definition is checked, then the (parent) compound module. If a compound module is subclassed before instantiated, the base type is checked first.

When no (non-default) assignment is found, the same places are searched in the *reverse order* for default value assignments. If no default value is found, an error may be raised or the user may be interactively prompted.

To illustrate the above rules, consider the following example where we want to assign parameter *p*:

```
simple A { double p; }
simple A2 extends A {...}
module B { submodules: a2: A2 {...} }
module B2 extends B {...}
network C { submodules: b2: B2 {...} }
```

Here, the search order is: *A, A2, a2, B, B2, b2, C*. NED conceptually searches the **parameters** blocks in that order for a (non-default) value, and then in reverse order for a default value.

The full search order and the form of assignment expected on each level:

1. `A { p = ...; }`
2. `A2 { p = ...; }`
3. `a2 { p = ...; }`
4. `B { a2.p = ...; }`
5. `B2 { a2.p = ...; }`
6. `b2 { a2.p = ...; }`
7. `C { b2.a2.p = ...; }`
8. `C { b2.a2.p = default(...); }`
9. `b2 { a2.p = default(...); }`
10. `B2 { a2.p = default(...); }`
11. `B { a2.p = default(...); }`
12. `a2 { p = default(...); }`
13. `A2 { p = default(...); }`
14. `A { p = default(...); }`

If only a default value is found or not even that, external configuration has a say. The configuration may contain an assignment for `C.b2.a2.p`; it may apply the default if there is one; it may ask the user interactively to enter a value; or if there is no default, it may raise an error *"no value for parameter"*.

A.4.18 Type Name Resolution

Names from other NED files can be referred to either by fully qualified name ("`inet.networklayer.ip.RoutingTable`"), or by short name ("`RoutingTable`") if the name is visible.

Visible names are:

- inner types of the same type or its super types;
- anything from the same package;
- imported names.

Imports

Imports have a similar syntax to Java, but they are more flexible with wildcards. All of the following are legal:

```
import inet.networklayer.ipv4.RoutingTable;
import inet.networklayer.ipv4.*;
import inet.networklayer.ipv4.Ro*Ta*;
import inet.*.ipv4.*;
import inet.**.RoutingTable;
```

One asterisk stands for any character sequence not containing dots; and a double asterisk stands for any character sequence (which may contain dots). No other wildcards are recognized.

An import not containing a wildcard must match an existing NED type. However, it is legal for an import that does contain wildcards not to match any NED type (although that might generate a warning.)

Inner types may not be referenced outside their enclosing types and their subclasses.

Base Types and Submodules

Fully qualified names and simple names are accepted. Simple names are looked up among the inner types of the enclosing type (compound module), then using imports, then in the same package.

Network Name in the Ini File

The network name in the ini file may be given as a fully qualified name or as a simple (unqualified) name.

Simple (unqualified) names are tried with the same package as the ini file is in (provided it is in a NED directory).

A.4.19 Resolution of Parametric Types

This section describes the type resolution for submodules and connections that are defined using the **`like`** keyword.

Type resolution is done in two steps. In the first step, the type name string expression is found and evaluated. Then in the second step, the resulting type name string is resolved to an actual NED type.

Step 1. The lookup of the type name string expression is similar to that of a parameter value lookup (A.4.17).

The expression may be specified locally (between the angle brackets), or using typename pattern assignments in any compound module that contains the submodule or connection directly or indirectly. A typename pattern is a pattern that ends in `.typename`.

Patterns are matched against the relative path of the submodule or connection, with `.typename` appended. The relative path is composed of a list of submodule names (name plus index) separated by dots; a connection is identified by the full name of its source gate plus the name of the channel object (which is currently always `channel`) separated by a dot.

> **NOTE:** As bidirectional connections are a shorthand for two unidirectional connections, the source gate name is qualified with `$i` or `$o` in the relative path.

An example that uses typename pattern assignment:

```
module Host {
    submodules:
        tcp: <> like ITCP;;
        ...
    connections:
        tcp.ipOut --> <> like IMyChannel --> ip.tcpIn;
}

network Network {
    parameters:
        host[*].tcp.typename = "TCP_lwIP";
        host[*].tcp.ipOut.channel.typename = "DebugChannel";
    submodules:
        host[10] : Host;
        ...
}
```

The general rules are the following:

1. A (non-default) parameter assignment may not be overridden later; that is, if there are assignments in multiple places, the assignment "closest" to the submodule or connection definition will be effective; others will be flagged as errors.
2. A default value is only used if a non-default value is not present. A non-default value may also come from a source external to NED, namely the simulation configuration (`omnetpp.ini`).
3. Unlike non-default values, a default value *may* be overridden; that is, if there are default value assignments in multiple places, the assignment "farthest" from the submodule or connection definition will win.
4. Among pattern assignments within the same **`parameters`** block, the first match will win. Patterns assignments with default and non-default values are considered to be two disjoint sets, only one of which are searched at a time.

This yields the following conceptual search order for typename assignments:

1. First, the submodule or connection definition is checked (angle brackets);
2. Then the compound module that contains the submodule or connection is checked for matching pattern assignments;
3. Then, assuming the compound module is part of a network, the search for matching pattern assignments continues up on the module tree until the root (the module that represents the network). At each level (compound module), first the specific submodule definition is checked, then the (parent) compound module. If a compound module is subclassed before instantiated, the base type is checked first.

When no (non-default) assignment is found, the same places are searched in the *reverse order* for default value assignments. If no default value is found, an error may be raised or the user may be interactively prompted.

To illustrate the above rules, consider the following example:

```
module A { submodules: h: <> like IFoo; }
module A2 extends A {...}
module B { submodules: a2: A2 {...} }
module B2 extends B {...}
network C { submodules: b2: B2 {...} }
```

Here, the search order is: *h, A, A2, a2, B, B2, b2, C*. NED conceptually searches the **parameters** blocks in that order for a (non-default) value, and then in reverse order for a default value.

The full search order and the form of assignment expected on each level:

1. `h:  <...> like IFoo;`
2. `A { h.typename = ...; }`
3. `A2 { h.typename = ...; }`
4. `a2 { h.typename = ...; }`
5. `B { a2.h.typename = ...; }`
6. `B2 { a2.h.typename = ...; }`
7. `b2 { a2.h.typename = ...; }`
8. `C { b2.a2.h.typename = ...; }`
9. `C { b2.a2.h.typename = default(...); }`
10. `b2 { a2.h.typename = default(...); }`
11. `B2 { a2.h.typename = default(...); }`
12. `B { a2.h.typename = default(...); }`
13. `a2 { h.typename = default(...); }`
14. `A2 { h.typename = default(...); }`

15. `A { h.typename = default(...); }`
16. `h: <default(...)> like IFoo;`

If only a default value is found or not even that, external configuration has a say. The configuration may contain an assignment for `C.b2.a2.h.typename`; it may apply the default value if there is one; it may ask the user interactively to enter a value; or if there is no default value, it may raise an error *"cannot determine submodule type"*.

Step 2. The type name string is expected to hold the simple name or fully qualified name of the desired NED type. Resolving the type name string to an actual NED type differs from normal type name lookups in that it ignores the imports in the file altogether. Instead, a list of NED types that have the given simple name or fully qualified name *and* implement the given interface is collected. The result must be exactly one module or channel type.

A.4.20 Implementing an Interface

A module type may implement one or more module interfaces, and a channel type may implement one or more channel interfaces, using the **`like`** keyword.

The module or channel type is required to have *at least* those parameters and gates that the interface has.

Regarding component properties, parameter properties and gate properties defined in the interface: the module or channel type is required to have at least the properties of the interface, with at least the same values. The component may have additional properties, and properties may add more keys and values.

> **NOTE:** Implementing an interface does not cause the properties, parameters and gates to be interited by the module or channel type; they have to be added explicitly.

> **NOTE:** A module or channel type may have extra properties, parameters and gates in addition to those in the interface.

A.4.21 Inheritance

Component inheritance is governed by the following rules:

- A simple module may only extend a simple module.
- A compound module may only extend a compound module.
- A channel may only extend a channel.
- A module interface may only extend a module interface (or several module interfaces).
- A channel interface may only extend a channel interface (or several channel interfaces).

A network is a shorthand for a compound module with the `@isNetwork` property set, so the same rules apply to it as to compound modules.

Inheritance may:

- add new properties, parameters, gates, inner types, submodules, connections, as long as names do not conflict with inherited names

- modify inherited properties, and properties of inherited parameters and gates
- it may not modify inherited submodules, connections and inner types

Other inheritance rules:

- for inner types: new inner types can be added, but inherited ones cannot be changed
- for properties: contents will be merged (rules like for display strings: values on same key and same position will overwrite old ones)
- for parameters: type cannot be redefined; value may be redefined in subclasses or at place of usage
- for gates: type cannot be redefined; vector size may be specified in subclasses or at place of usage if it was unspecified
- for gate/parameter properties: extra properties can be added; existing properties can be overridden/extended as for standalone properties
- for submodules: new submodules may be added, but inherited ones cannot be modified
- for connections: new connections may be added, but inherited ones cannot be modified

The following sections will elaborate on the above rules.

Property Inheritance

Generally, properties may be modified via inheritance. Inheritance may:

- add new keys
- add/overwrite values for existing keys
- remove a value from an existing key by using hyphen as a special value

Parameter Inheritance

Default values for parameters may be overridden in subclasses.

Gate Inheritance

Gate vector size may not be overridden in subclasses.

A.4.22 Network Build Order

When a network is instantiated for simulation, the module tree is built in a top-down preorder fashion. This means that starting from an empty system module, all submodules are created, their parameters and vector sizes are assigned, and they get fully connected before proceeding to go into the submodules to build their internals.

This implies that inside a compound module definition (including in submodules and connections), one can refer to the compound module's parameters and gate sizes, because they are already built at the time of usage.

The same rules apply to compound or simple modules created dynamically during runtime.

A.5 Expressions

NED language expressions have a C-like syntax, with some variations on operator names (see `^`, `#`, `##`). Expressions may refer to module parameters, loop variables (inside connection **`for`** loops), gate vector and module vector sizes, and other attributes of the model. Expressions can use built-in and user-defined functions as well.

NOTE: New NED functions can be defined in C++; see section 7.11.

A.5.1 Operators

The following operators are supported (in order of decreasing precedence):

Operator	Meaning
`-`, `!`, ~	unary minus, negation, bitwise complement
`^`	power-of
`*`, `/`, `%`	multiply, divide, integer modulo
`+`, `-`	add, subtract, string concatenation
«, »	bitwise shift
`&`, `|`, `#`	bitwise and, or, xor

`==`
`!=`
`>`, `>=`
`<`, `<=`
	equal
	not equal
	greater than, greater than or equal to
	less than, less than or equal to

`&&`, `||`, `##`	logical operators and, or, xor
`?:`	the C/C++ "inline if"

Conversions

Values may have the same types as NED parameters: boolean, integer, double, string, or XML element. An integer or double value may have an associated measurement unit (`s`, `mW`, etc.)

Integer and double values are implicitly converted to one another where needed.

There is no implicit conversion between boolean and numeric types, so `0` is not a synonym for **`false`**, and nonzero numbers are not a synonym for **`true`**.

There is also no conversion between string and numeric types, so e.g. `"foo"+5` is illegal. There are functions for converting a number to string and vice versa.

For bitwise operators and modulo (%), doubles are converted to integers.

NOTE: Integers are represented with the C++ type `long`. `double`-to-`long` conversions are performed using the C++ language's built-in typecast operator. As `long` is 32 bits on most 32-bit architectures, for portability reasons it is not recommended to use integers that do not fit into 32 bits.

Unit Handling

Operations involving numbers with units work in the following way:

Addition, subtraction, and numeric comparisons require their arguments to have the same

unit or compatible units; in the latter case a unit conversion is performed before the operation. Incompatible units cause an error.

Modulo, power-of and the bitwise operations require their arguments to be dimensionless, otherwise the result would depend on the choice of the unit.

> **NOTE:** If you need a floating-point modulo operator that handles units as well, use the `fmod()` function.

Multiplying two numbers with units is not supported.

For division, dividing two numbers with units is only supported if the two units are convertible (i.e. the result will be dimensionless). Dividing a dimensionless number with a number with unit is not supported.

A.5.2 Referencing Parameters and Loop Variables

Identifiers in expressions occurring *anywhere* in component definitions are interpreted as referring to parameters of the given component. For example, identifiers inside submodule bodies refer to the parameters of the compound module.

Expressions may also refer to parameters of submodules defined earlier in the NED file, using the `submoduleName.paramName` or the `submoduleName[index].paramName` syntax. To refer to parameters of the local submodule inside a submodule body, use the **`this`** qualifier: `this.destAddress`.

Exception: if an identifier occurs in a connection **`for`** loop and names a previously defined loop variable, then it is understood as referring to the loop variable.

A.5.3 The `index` Operator

The **`index`** operator is only allowed in a vector submodule's body, and yields the index of the submodule instance.

A.5.4 The `sizeof()` Operator

The **`sizeof()`** operator expects one argument, and it is only accepted in compound module definitions.

The `sizeof(identifier)` syntax occurring *anywhere* in a compound module yields the size of the named submodule or gate vector of the compound module.

Inside submodule bodies, the size of a gate vector of the same submodule can be referred to with the **`this`** qualifier: `sizeof(this.out)`.

To refer to the size of a submodule's gate vector defined earlier in the NED file, use the `sizeof(submoduleName.gateVectorName)` or `sizeof(submoduleName[index].gateVectorName)` syntax.

A.5.5 Functions

The functions available in NED are listed in Appendix D.

Selected functions are documented below.

The `xmldoc()` Function

The **`xmldoc()`** NED function can be used to assign **`xml`** parameters, that is, point them to XML files or to specific elements inside XML files.

`xmldoc()` accepts a file name as well as an optional second string argument that contains an XPath-like expression.

The XPath expression is used to select an element within the document. If the expression matches several elements, the first element (in preorder depth-first traversal) will be selected. (This is unlike XPath, which selects all matching nodes.)

The expression syntax is the following:

- An expression consists of *path components* (or "steps") separated by "`/`" or "`//`".
- A path component can be an element tag name, "`*`", "`.`" or "`..`".
- "`/`" means child element (just as in `/usr/bin/gcc`); "`//`" means an element any number of levels under the current element.
- "`.`", "`..`" and "`*`" mean the current element, the parent element, and an element with any tag name, respectively.
- Element tag names and "`*`" can have an optional predicate in the form "`[position]`" or "`[@attribute='value']`". Positions start from zero.
- Predicates of the form "`[@attribute=`*`$param`*`]`" are also accepted, where *`$param`* can be one of: `$MODULE_FULLPATH`, `$MODULE_FULLNAME`, `$MODULE_NAME`, `$MODULE_INDEX`, `$MODULE_ID`, `$PARENTMODULE_FULLPATH`, `$PARENTMODULE_FULLNAME`, `$PARENTMODULE_NAME`, `$PARENTMODULE_INDEX`, `$PARENTMODULE_ID`, `$GRANDPARENTMODULE_FULLPATH`, `$GRANDPARENTMODULE_FULLNAME`, `$GRANDPARENTMODULE_NAME`, `$GRANDPARENTMODULE_INDEX`, `$GRANDPARENTMODULE_ID`.

The `xml()` Function

The **`xml()`** NED function can be used to parse a string as an XML document, and assign the result to an **`xml`** parameter.

`xml()` accepts the string to be parsed as well as an optional second string argument that contains an XPath-like expression.

The XPath expression is used in the same manner as with the **`xmldoc()`** function.

A.5.6 Units of Measurement

The following measurements units are recognized in constants. Other units can be used as well, but there are no conversions available for them (i.e. `parsec` and `kiloparsec` will be treated as two completely unrelated units.)

Unit	Name	Value
s	second	
d	day	86400s
h	hour	3600s
min	minute	60s

ms	millisecond	1e-3s
us	microsecond	1e-6s
ns	nanosecond	1e-9s
ps	picosecond	1e-12s
bps	bit/sec	
kbps	kilobit/sec	1e3bps
Mbps	megabit/sec	1e6bps
Gbps	gigabit/sec	1e9bps
Tbps	terabit/sec	1e12bps
B	byte	
KiB	kilo (kibi) byte	1024B
MiB	mega (mebi) byte	1.04858e6B
GiB	giga (gibi) byte	1.07374e9B
TiB	tera (tebi) byte	1.09951e12B
b	bit	
m	meter	
km	kilometer	1e3m
cm	centimeter	1e-2m
mm	millimeter	1e-3m
W	watt	
mW	milliwatt	1e-3W
Hz	hertz	
kHz	kilohertz	1e3Hz
MHz	megahertz	1e6Hz
GHz	gigahertz	1e9Hz
g	gram	1e-3kg
kg	kilogram	
J	joule	
kJ	kilojoule	1e3J
MJ	megajoule	1e6J
V	volt	
kV	kilovolt	1e3V
mV	millivolt	1e-3V
A	ampere	
mA	milliampere	1e-3A
uA	microampere	1e-6A
mps	meter/sec	
kmph	kilometer/hour	(1/3.6)mps

Appendix B

NED Language Grammar

This appendix contains the grammar for the NED language.

In the NED language, space, horizontal tab and new line characters count as delimiters, so one or more of them is required between two elements of the description which would otherwise be unseparable.

'//' (two slashes) may be used to write comments that last to the end of the line.

The language is fully case sensitive.

Notation:

- rule syntax is that of *bison/yacc*
- uppercase words are terminals, lowercase words are nonterminals
- `NAME`, `STRINGCONSTANT`, `INTCONSTANT`, `REALCONSTANT` represent identifier names and string, integer and real number literals (defined as in the C language, except that a `0` prefix does not stand for octal notation)
- other terminals represent keywords in all lowercase

```
nedfile
        : definitions
        |
        ;

definitions
        : definitions definition
        | definition
        ;

definition
        : packagedeclaration
        | import
        | propertydecl
        | fileproperty
        | channeldefinition
        | channelinterfacedefinition
        | simplemoduledefinition
```

```
        | compoundmoduledefinition
        | networkdefinition
        | moduleinterfacedefinition
        | ';'
        ;

packagedeclaration
        : PACKAGE dottedname ';'
        ;

dottedname
        : dottedname '.' NAME
        | NAME
        ;

import
        : IMPORT importspec ';'
        ;

importspec
        : importspec '.' importname
        | importname
        ;

importname
        : importname NAME
        | importname '*'
        | importname '**'
        | NAME
        | '*'
        | '**'
        ;

propertydecl
        : propertydecl_header opt_inline_properties ';'
        | propertydecl_header '(' opt_propertydecl_keys ')' opt_inline_properties ';'
        ;

propertydecl_header
        : PROPERTY '@' PROPNAME
        | PROPERTY '@' PROPNAME '[' ']'
        ;

opt_propertydecl_keys
        : propertydecl_keys
        |
        ;

propertydecl_keys
        : propertydecl_keys ';' propertydecl_key
        | propertydecl_key
        ;
```

```
propertydecl_key
        : property_literal
        ;

fileproperty
        : property_namevalue ';'
        ;

channeldefinition
        : channelheader '{'
            opt_paramblock
          '}'
        ;

channelheader
        : CHANNEL NAME
           opt_inheritance
        ;

opt_inheritance
        :
        | EXTENDS extendsname
        | LIKE likenames
        | EXTENDS extendsname LIKE likenames
        ;

extendsname
        : dottedname
        ;

likenames
        : likenames ',' likename
        | likename
        ;

likename
        : dottedname
        ;

channelinterfacedefinition
        : channelinterfaceheader '{'
            opt_paramblock
          '}'
        ;

channelinterfaceheader
        : CHANNELINTERFACE NAME
           opt_interfaceinheritance
        ;

opt_interfaceinheritance
```

```
        : EXTENDS extendsnames
        |
        ;

extendsnames
        : extendsnames ',' extendsname
        | extendsname
        ;

simplemoduledefinition
        : simplemoduleheader '{'
            opt_paramblock
            opt_gateblock
          '}'
        ;

simplemoduleheader
        : SIMPLE NAME
          opt_inheritance
        ;

compoundmoduledefinition
        : compoundmoduleheader '{'
            opt_paramblock
            opt_gateblock
            opt_typeblock
            opt_submodblock
            opt_connblock
          '}'
        ;

compoundmoduleheader
        : MODULE NAME
          opt_inheritance
        ;

networkdefinition
        : networkheader '{'
            opt_paramblock
            opt_gateblock
            opt_typeblock
            opt_submodblock
            opt_connblock
          '}'
        ;

networkheader
        : NETWORK NAME
          opt_inheritance
        ;

moduleinterfacedefinition
```

```
        : moduleinterfaceheader '{'
            opt_paramblock
            opt_gateblock
          '}'
        ;

moduleinterfaceheader
        : MODULEINTERFACE NAME
           opt_interfaceinheritance
        ;

opt_paramblock
        : opt_params
        | PARAMETERS ':'
          opt_params
        ;

opt_params
        : params
        |
        ;

params
        : params paramsitem
        | paramsitem
        ;

paramsitem
        : param
        | property
        ;

param
        : param_typenamevalue
        | pattern_value
        ;

param_typenamevalue
        : param_typename opt_inline_properties ';'
        | param_typename opt_inline_properties '=' paramvalue opt_inline_properties ';'
        ;

param_typename
        : opt_volatile paramtype NAME
        | NAME
        ;

pattern_value
        : pattern '=' paramvalue ';'
        ;

paramtype
```

```
        : DOUBLE
        | INT
        | STRING
        | BOOL
        | XML
        ;

opt_volatile
        : VOLATILE
        |
        ;

paramvalue
        : expression
        | DEFAULT '(' expression ')'
        | DEFAULT
        | ASK
        ;

opt_inline_properties
        : inline_properties
        |
        ;

inline_properties
        : inline_properties property_namevalue
        | property_namevalue
        ;

pattern
        : pattern2 '.' pattern_elem
        | pattern2 '.' TYPENAME
        ;

pattern2
        : pattern2 '.' pattern_elem
        | pattern_elem
        ;

pattern_elem
        : pattern_name
        | pattern_name '[' pattern_index ']'
        | pattern_name '[' '*' ']'
        | '**'
        ;

pattern_name
        : NAME
        | NAME '$' NAME
        | CHANNEL
        | '{' pattern_index '}'
        | '*'
```

```
        | pattern_name NAME
        | pattern_name '{' pattern_index '}'
        | pattern_name '*'
        ;

pattern_index
        : INTCONSTANT
        | INTCONSTANT '..' INTCONSTANT
        | '..' INTCONSTANT
        | INTCONSTANT '..'
        ;

property
        : property_namevalue ';'
        ;

property_namevalue
        : property_name
        | property_name '(' opt_property_keys ')'
        ;

property_name
        : '@' PROPNAME
        | '@' PROPNAME '[' PROPNAME ']'
        ;

opt_property_keys
        : property_keys
        ;

property_keys
        : property_keys ';' property_key
        | property_key
        ;

property_key
        : property_literal '=' property_values
        | property_values
        ;

property_values
        : property_values ',' property_value
        | property_value
        ;

property_value
        : property_literal
        |
        ;

property_literal
        : property_literal CHAR
```

```
        | property_literal STRINGCONSTANT
        | CHAR
        | STRINGCONSTANT
        ;

opt_gateblock
        : gateblock
        |
        ;

gateblock
        : GATES ':'
          opt_gates
        ;

opt_gates
        : gates
        |
        ;

gates
        : gates gate
        | gate
        ;

gate
        : gate_typenamesize
          opt_inline_properties ';'
        ;

gate_typenamesize
        : gatetype NAME
        | gatetype NAME '[' ']'
        | gatetype NAME vector
        | NAME
        | NAME '[' ']'
        | NAME vector
        ;

gatetype
        : INPUT
        | OUTPUT
        | INOUT
        ;

opt_typeblock
        : typeblock
        |
        ;

typeblock
        : TYPES ':'
```

```
           opt_localtypes
        ;

opt_localtypes
        : localtypes
        |
        ;

localtypes
        : localtypes localtype
        | localtype
        ;

localtype
        : propertydecl
        | channeldefinition
        | channelinterfacedefinition
        | simplemoduledefinition
        | compoundmoduledefinition
        | networkdefinition
        | moduleinterfacedefinition
        | ';'
        ;

opt_submodblock
        : submodblock
        |
        ;

submodblock
        : SUBMODULES ':'
          opt_submodules
        ;

opt_submodules
        : submodules
        |
        ;

submodules
        : submodules submodule
        | submodule
        ;

submodule
        : submoduleheader ';'
        | submoduleheader '{'
          opt_paramblock
          opt_gateblock
          '}' opt_semicolon
        ;
```

```
submoduleheader
        : submodulename ':' dottedname opt_condition
        | submodulename ':' likeexpr LIKE dottedname opt_condition
        ;

submodulename
        : NAME
        |  NAME vector
        ;

likeexpr
        : '<' '>'
        | '<' expression '>'
        | '<' DEFAULT '(' expression ')' '>'
        ;

opt_condition
        : condition
        |
        ;

opt_connblock
        : connblock
        |
        ;

connblock
        : CONNECTIONS ALLOWUNCONNECTED ':'
          opt_connections
        | CONNECTIONS ':'
          opt_connections
        ;

opt_connections
        : connections
        |
        ;

connections
        : connections connectionsitem
        | connectionsitem
        ;

connectionsitem
        : connectiongroup
        | connection opt_loops_and_conditions ';'
        ;

connectiongroup
        : opt_loops_and_conditions '{'
          connections '}' opt_semicolon
        ;
```

```
opt_loops_and_conditions
        : loops_and_conditions
        |
        ;

loops_and_conditions
        : loops_and_conditions ',' loop_or_condition
        | loop_or_condition
        ;

loop_or_condition
        : loop
        | condition
        ;

loop
        : FOR NAME '=' expression '..' expression
        ;

connection
        : leftgatespec '-->' rightgatespec
        | leftgatespec '-->' channelspec '-->' rightgatespec
        | leftgatespec '<--' rightgatespec
        | leftgatespec '<--' channelspec '<--' rightgatespec
        | leftgatespec '<-->' rightgatespec
        | leftgatespec '<-->' channelspec '<-->' rightgatespec
        ;

leftgatespec
        : leftmod '.' leftgate
        | parentleftgate
        ;

leftmod
        : NAME vector
        | NAME
        ;

leftgate
        : NAME opt_subgate
        | NAME opt_subgate vector
        | NAME opt_subgate '++'
        ;

parentleftgate
        : NAME opt_subgate
        | NAME opt_subgate vector
        | NAME opt_subgate '++'
        ;

rightgatespec
```

```
        : rightmod '.' rightgate
        | parentrightgate
        ;

rightmod
        : NAME
        | NAME vector
        ;

rightgate
        : NAME opt_subgate
        | NAME opt_subgate vector
        | NAME opt_subgate '++'
        ;

parentrightgate
        : NAME opt_subgate
        | NAME opt_subgate vector
        | NAME opt_subgate '++'
        ;

opt_subgate
        : '$' NAME
        |
        ;

channelspec
        : channelspec_header
        | channelspec_header '{'
            opt_paramblock
          '}'
        ;

channelspec_header
        : opt_channelname
        | opt_channelname dottedname
        | opt_channelname likeexpr LIKE dottedname
        ;

opt_channelname
        :
        | NAME ':'
        ;

condition
        : IF expression
        ;

vector
        : '[' expression ']'
        ;
```

```
expression
        :
          expr
        ;

expr
        : simple_expr
        | '(' expr ')'
        | CONST '(' expr ')'
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr '%' expr
        | expr '^' expr
        | '-' expr
        | expr '==' expr
        | expr '!=' expr
        | expr '>' expr
        | expr '>=' expr
        | expr '<' expr
        | expr '<=' expr
        | expr '&&' expr
        | expr '||' expr
        | expr '##' expr
        | '!' expr
        | expr '&' expr
        | expr '|' expr
        | expr '#' expr
        | '~' expr
        | expr '<<' expr
        | expr '>>' expr
        | expr '?' expr ':' expr
        | INT '(' expr ')'
        | DOUBLE '(' expr ')'
        | STRING '(' expr ')'
        | funcname '(' ')'
        | funcname '(' expr ')'
        | funcname '(' expr ',' expr ')'
        | funcname '(' expr ',' expr ',' expr ')'
        | funcname '(' expr ',' expr ',' expr ',' expr ')'
        | funcname '(' expr ',' expr ',' expr ',' expr ',' expr ')'
        | funcname '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ')'
        | funcname '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ')'
        | funcname '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' ex
        | funcname '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' ex
        | funcname '(' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' expr ',' ex
        ;

simple_expr
        : identifier
        | special_expr
```

```
        | literal
        ;

funcname
        : NAME
        | XMLDOC
        | XML
        ;

identifier
        : NAME
        | THIS '.' NAME
        | NAME '.' NAME
        | NAME '[' expr ']' '.' NAME
        ;

special_expr
        : INDEX
        | INDEX '(' ')'
        | SIZEOF '(' identifier ')'
        ;

literal
        : stringliteral
        | boolliteral
        | numliteral
        ;

stringliteral
        : STRINGCONSTANT
        ;

boolliteral
        : TRUE
        | FALSE
        ;

numliteral
        : INTCONSTANT
        | REALCONSTANT
        | quantity
        ;

quantity
        : quantity INTCONSTANT NAME
        | quantity REALCONSTANT NAME
        | INTCONSTANT NAME
        | REALCONSTANT NAME
        ;

opt_semicolon
        : ';'
```

```
        |
        ;
```

Appendix C

NED XML Binding

This appendix shows the DTD for the XML binding of the NED language and message definitions.

```
<!ELEMENT files ((ned-file|msg-file)*)>

<!--
  **  NED-2.
  -->

<!ELEMENT ned-file (comment*, (package|import|property-decl|property|
                    simple-module|compound-module|module-interface|
                    channel|channel-interface)*)>
<!ATTLIST ned-file
     filename           CDATA     #REQUIRED
     version            CDATA     "2">

<!-- comments and whitespace; comments include '//' marks. Note that although
     nearly all elements may contain comment elements, there are places
     (e.g. within expressions) where they are ignored by the implementation.
     Default value is a space or a newline, depending on the context.
 -->
<!ELEMENT comment EMPTY>
<!ATTLIST comment
     locid              NMTOKEN   #REQUIRED
     content            CDATA     #IMPLIED>

<!ELEMENT package (comment*)>
<!ATTLIST package
     name               CDATA     #REQUIRED>

<!ELEMENT import (comment*)>
<!ATTLIST import
     import-spec        CDATA     #REQUIRED>

<!ELEMENT property-decl (comment*, property-key*, property*)>
<!ATTLIST property-decl
```

```
     name               NMTOKEN   #REQUIRED
     is-array           (true|false) "false">

<!ELEMENT extends (comment*)>
<!ATTLIST extends
     name               CDATA     #REQUIRED>

<!ELEMENT interface-name (comment*)>
<!ATTLIST interface-name
     name               CDATA     #REQUIRED>

<!ELEMENT simple-module (comment*, extends?, interface-name*, parameters?, gates?)>
<!ATTLIST simple-module
     name               NMTOKEN   #REQUIRED>

<!ELEMENT module-interface (comment*, extends*, parameters?, gates?)>
<!ATTLIST module-interface
     name               NMTOKEN   #REQUIRED>

<!ELEMENT compound-module (comment*, extends?, interface-name*,
                           parameters?, gates?, types?, submodules?, connections?)>
<!ATTLIST compound-module
     name               NMTOKEN   #REQUIRED>

<!ELEMENT channel-interface (comment*, extends*, parameters?)>
<!ATTLIST channel-interface
     name                NMTOKEN   #REQUIRED>

<!ELEMENT channel (comment*, extends?, interface-name*, parameters?)>
<!ATTLIST channel
     name                NMTOKEN   #REQUIRED>

<!ELEMENT parameters (comment*, (property|param)*)>
<!ATTLIST parameters
    is-implicit         (true|false)  "false">

<!ELEMENT param (comment*, expression?, property*)>
<!ATTLIST param
     type               (double|int|string|bool|xml) #IMPLIED
     is-volatile        (true|false)  "false"
     name               CDATA     #REQUIRED
     value              CDATA     #IMPLIED
     is-pattern         (true|false)  "false"
     is-default         (true|false)  "false">

<!ELEMENT property (comment*, property-key*)>
<!ATTLIST property
     is-implicit        (true|false) "false"
     name               NMTOKEN   #REQUIRED
     index              NMTOKEN   #IMPLIED>

<!ELEMENT property-key (comment*, literal*)>
```

```
<!ATTLIST property-key
     name               CDATA     #IMPLIED>

<!ELEMENT gates (comment*, gate*)>

<!ELEMENT gate (comment*, expression?, property*)>
<!ATTLIST gate
     name               NMTOKEN   #REQUIRED
     type               (input|output|inout) #IMPLIED
     is-vector          (true|false) "false"
     vector-size        CDATA     #IMPLIED>

<!ELEMENT types (comment*, (channel|channel-interface|simple-module|
                                compound-module|module-interface)*)>

<!ELEMENT submodules (comment*, submodule*)>

<!ELEMENT submodule (comment*, expression*, condition?, parameters?, gates?)>
<!ATTLIST submodule
     name               NMTOKEN   #REQUIRED
     type               CDATA     #IMPLIED
     like-type          CDATA     #IMPLIED
     like-expr          CDATA     #IMPLIED
     is-default         (true|false) "false"
     vector-size        CDATA     #IMPLIED>

<!ELEMENT connections (comment*, (connection|connection-group)*)>
<!ATTLIST connections
     allow-unconnected (true|false) "false">

<!ELEMENT connection (comment*, expression*, parameters?, (loop|condition)*)>
<!ATTLIST connection
     src-module          NMTOKEN   #IMPLIED
     src-module-index    CDATA     #IMPLIED
     src-gate            NMTOKEN   #REQUIRED
     src-gate-plusplus   (true|false) "false"
     src-gate-index      CDATA     #IMPLIED
     src-gate-subg       (i|o)     #IMPLIED
     dest-module         NMTOKEN   #IMPLIED
     dest-module-index   CDATA     #IMPLIED
     dest-gate           NMTOKEN   #REQUIRED
     dest-gate-plusplus (true|false) "false"
     dest-gate-index     CDATA     #IMPLIED
     dest-gate-subg      (i|o)     #IMPLIED
     name                NMTOKEN   #IMPLIED
     type                CDATA     #IMPLIED
     like-type           CDATA     #IMPLIED
     like-expr           CDATA     #IMPLIED
     is-default         (true|false) "false"
     is-bidirectional   (true|false) "false"
     is-forward-arrow   (true|false) "true">
```

```
<!ELEMENT connection-group (comment*, (loop|condition)*, connection*)>

<!ELEMENT loop (comment*, expression*)>
<!ATTLIST loop
     param-name          NMTOKEN   #REQUIRED
     from-value          CDATA     #IMPLIED
     to-value            CDATA     #IMPLIED>

<!ELEMENT condition (comment*, expression?)>
<!ATTLIST condition
     condition           CDATA     #IMPLIED>

<!--
  **  Expressions
  -->

<!ELEMENT expression (comment*, (operator|function|ident|literal))>
<!ATTLIST expression
     target              CDATA     #IMPLIED>

<!ELEMENT operator (comment*, (operator|function|ident|literal)+)>
<!ATTLIST operator
     name                CDATA     #REQUIRED>

<!-- functions, "index", "const" and "sizeof" -->
<!ELEMENT function (comment*, (operator|function|ident|literal)*)>
<!ATTLIST function
     name                NMTOKEN   #REQUIRED>

<!-- Ident is either a parameter reference or an argument for the "sizeof"
     operator; its NED source form is one of the following: "name", "module.name",
     "module[n].name". If there is a child, it represents the module index n.
     A reserved module name (with special meaning) is "this".
-->
<!ELEMENT ident (comment*, (operator|function|ident|literal)?)>
<!ATTLIST ident
     module              CDATA     #IMPLIED
     name                NMTOKEN   #REQUIRED>


<!ELEMENT literal (comment*)>
<!-- Note: value is in fact REQUIRED, but empty attr value should
     also be accepted because that represents the "" string literal;
     "spec" is for properties, to store the null value and "-",
     the antivalue. Unit can only be present with "double".
 -->
<!ATTLIST literal
     type  (double|int|string|bool|spec)  #REQUIRED
     unit                CDATA     #IMPLIED
     text                CDATA     #IMPLIED
     value               CDATA     #IMPLIED>
```

```
<!--================================================================-->

<!--
**
** OMNeT++/OMNEST Message Definitions (MSG)
**
-->

<!ELEMENT msg-file (comment*, (namespace|property-decl|property|cplusplus|
                    struct-decl|class-decl|message-decl|packet-decl|enum-decl|
                    struct|class|message|packet|enum)*)>

<!ATTLIST msg-file
     filename            CDATA     #IMPLIED
     version             CDATA     "2">

<!ELEMENT namespace (comment*)>
<!ATTLIST namespace
     name                CDATA     #REQUIRED>  <!-- note: not NMTOKEN because it may co

<!ELEMENT cplusplus (comment*)>
<!ATTLIST cplusplus
     body                CDATA     #REQUIRED>

<!-- C++ type announcements -->

<!ELEMENT struct-decl (comment*)>
<!ATTLIST struct-decl
     name                CDATA     #REQUIRED>

<!ELEMENT class-decl (comment*)>
<!ATTLIST class-decl
     name                CDATA     #REQUIRED
     is-cobject      (true|false)  "false"
     extends-name        CDATA     #IMPLIED>

<!ELEMENT message-decl (comment*)>
<!ATTLIST message-decl
     name                CDATA     #REQUIRED>

<!ELEMENT packet-decl (comment*)>
<!ATTLIST packet-decl
     name                CDATA     #REQUIRED>

<!ELEMENT enum-decl (comment*)>
<!ATTLIST enum-decl
     name                CDATA     #REQUIRED>

<!-- Enums -->

<!ELEMENT enum (comment*, enum-fields?)>
<!ATTLIST enum
```

```
     name                NMTOKEN   #REQUIRED
     extends-name        CDATA     #IMPLIED
     source-code         CDATA     #IMPLIED>

<!ELEMENT enum-fields (comment*, enum-field*)>

<!ELEMENT enum-field (comment*)>
<!ATTLIST enum-field
     name                NMTOKEN   #REQUIRED
     value               CDATA     #IMPLIED>

<!-- Message, class, struct -->

<!ELEMENT message (comment*, (property|field)*)>
<!ATTLIST message
     name                NMTOKEN   #REQUIRED
     extends-name        CDATA     #IMPLIED
     source-code         CDATA     #IMPLIED>

<!ELEMENT packet (comment*, (property|field)*)>
<!ATTLIST packet
     name                NMTOKEN   #REQUIRED
     extends-name        CDATA     #IMPLIED
     source-code         CDATA     #IMPLIED>

<!ELEMENT class (comment*, (property|field)*)>
<!ATTLIST class
     name                NMTOKEN   #REQUIRED
     extends-name        CDATA     #IMPLIED
     source-code         CDATA     #IMPLIED>

<!ELEMENT struct (comment*, (property|field)*)>
<!ATTLIST struct
     name                NMTOKEN   #REQUIRED
     extends-name        CDATA     #IMPLIED
     source-code         CDATA     #IMPLIED>

<!ELEMENT field (comment*, property*)>
<!ATTLIST field
     name                NMTOKEN   #REQUIRED
     data-type           CDATA     #IMPLIED
     is-abstract     (true|false)  "false"
     is-readonly     (true|false)  "false"
     is-vector       (true|false)  "false"
     vector-size         CDATA     #IMPLIED
     default-value       CDATA     #IMPLIED>

<!--
  **  'unknown' is used internally to represent elements not in this NED DTD
  -->
<!ELEMENT unknown        ANY>
<!ATTLIST unknown
```

```
    element             CDATA     #REQUIRED>
```

Appendix D

NED Functions

The functions that can be used in NED expressions and ini files are the following. The question mark (as in "`rng?`") marks optional arguments.

Category "conversion":

double : `double double(any x)`
Converts x to double, and returns the result. A boolean argument becomes 0 or 1; a string is interpreted as number; an XML argument causes an error.

int : `int int(any x)`
Converts x to an integer (C++ long), and returns the result. A boolean argument becomes 0 or 1; a double is converted using floor(); a string is interpreted as number; an XML argument causes an error.

string : `string string(any x)`
Converts x to string, and returns the result.

Category "math":

acos : `double acos(double)`
Trigonometric function; see standard C function of the same name

asin : `double asin(double)`
Trigonometric function; see standard C function of the same name

atan : `double atan(double)`
Trigonometric function; see standard C function of the same name

atan2 : `double atan2(double, double)`
Trigonometric function; see standard C function of the same name

ceil : `double ceil(double)`
Rounds down; see standard C function of the same name

cos : `double cos(double)`
Trigonometric function; see standard C function of the same name

exp : `double exp(double)`
Exponential; see standard C function of the same name

fabs : `quantity fabs(quantity x)`
Returns the absolute value of the quantity.

floor : `double floor(double)`
Rounds up; see standard C function of the same name

fmod : `quantity fmod(quantity x, quantity y)`
Returns the floating-point remainder of x/y; unit conversion takes place if needed.

hypot : `double hypot(double, double)`
Length of the hypotenuse; see standard C function of the same name

log : `double log(double)`
Natural logarithm; see standard C function of the same name

log10 : `double log10(double)`
Base-10 logarithm; see standard C function of the same name

max : `quantity max(quantity a, quantity b)`
Returns the greater one of the two quantities; unit conversion takes place if needed.

min : `quantity min(quantity a, quantity b)`
Returns the smaller one of the two quantities; unit conversion takes place if needed.

pow : `double pow(double, double)`
Power; see standard C function of the same name

sin : `double sin(double)`
Trigonometric function; see standard C function of the same name

sqrt : `double sqrt(double)`
Square root; see standard C function of the same name

tan : `double tan(double)`
Trigonometric function; see standard C function of the same name

Category "misc":

firstAvailable : `string firstAvailable(...)`
Accepts any number of strings, interprets them as NED type names (qualified or unqualified), and returns the first one that exists and its C++ implementation class is also available. Throws an error if none of the types are available.

simTime : `quantity simTime()`
Returns the current simulation time.

select : `any select(int index, ...)`
Returns the <index>th item from the rest of the argument list; numbering starts from 0.

Category "ned":

ancestorIndex : `int ancestorIndex(int numLevels)`
Returns the index of the ancestor module numLevels levels above the module or channel in context.

fullName : `string fullName()`
Returns the full name of the module or channel in context.

fullPath : `string fullPath()`
Returns the full path of the module or channel in context.

parentIndex : `int parentIndex()`
Returns the index of the parent module, which has to be part of module vector.

Category "random/continuous":

beta : `double beta(double alpha1, double alpha2, int rng?)`
Returns a random number from the Beta distribution

cauchy : `quantity cauchy(quantity a, quantity b, int rng?)`
Returns a random number from the Cauchy distribution

chi_square : `double chi_square(int k, int rng?)`
Returns a random number from the Chi-square distribution

erlang_k : `quantity erlang_k(int k, quantity mean, int rng?)`
Returns a random number from the Erlang distribution

exponential : `quantity exponential(quantity mean, int rng?)`
Returns a random number from the Exponential distribution

gamma_d : `quantity gamma_d(double alpha, quantity theta, int rng?)`
Returns a random number from the Gamma distribution

lognormal : `double lognormal(double m, double w, int rng?)`
Returns a random number from the Lognormal distribution

normal : `quantity normal(quantity mean, quantity stddev, int rng?)`
Returns a random number from the Normal distribution

pareto_shifted : `quantity pareto_shifted(double a, quantity b, quantity c, int rng?)`
Returns a random number from the Pareto-shifted distribution

student_t : `double student_t(int i, int rng?)`
Returns a random number from the Student-t distribution

triang : `quantity triang(quantity a, quantity b, quantity c, int rng?)`
Returns a random number from the Triangular distribution

truncnormal : `quantity truncnormal(quantity mean, quantity stddev, int rng?)`
Returns a random number from the truncated Normal distribution

uniform : `quantity uniform(quantity a, quantity b, int rng?)`
Returns a random number from the Uniform distribution

weibull : `quantity weibull(quantity a, quantity b, int rng?)`
Returns a random number from the Weibull distribution

Category "random/discrete":

bernoulli : `int bernoulli(double p, int rng?)`
Returns a random number from the Bernoulli distribution

binomial : `int binomial(int n, double p, int rng?)`
Returns a random number from the Binomial distribution

geometric : `int geometric(double p, int rng?)`
Returns a random number from the Geometric distribution

intuniform : `int intuniform(int a, int b, int rng?)`
Returns a random number from the Intuniform distribution

negbinomial : `int negbinomial(int n, double p, int rng?)`
Returns a random number from the Negbinomial distribution

poisson : `int poisson(double lambda, int rng?)`
Returns a random number from the Poisson distribution

Category "strings":

choose : `string choose(int index, string list)`
Interprets list as a space-separated list, and returns the item at the given index. Negative and out-of-bounds indices cause an error.

contains : `bool contains(string s, string substr)`
Returns true if string s contains substr as substring

endsWith : `bool endsWith(string s, string substr)`
Returns true if s ends with the substring substr.

expand : `string expand(string s)`
Expands $ variables ($configname, $runnumber, etc.) in the given string, and returns the result.

indexOf : `int indexOf(string s, string substr)`
Returns the position of the first occurrence of substring substr in s, or -1 if s does not contain substr.

length : `int length(string s)`
Returns the length of the string

replace : `string replace(string s, string substr, string repl, int startPos?)`
Replaces all occurrences of substr in s with the string repl. If startPos is given, search begins from position startPos in s.

replaceFirst : `string replaceFirst(string s, string substr, string repl, int startPos?)`
Replaces the first occurrence of substr in s with the string repl. If startPos is given, search begins from position startPos in s.

startsWith : `bool startsWith(string s, string substr)`
Returns true if s begins with the substring substr.

substring : `string substring(string s, int pos, int len?)`
Return the substring of s starting at the given position, either to the end of the string or maximum len characters

substringAfter : `string substringAfter(string s, string substr)`
Returns the substring of s after the first occurrence of substr, or the empty string if s does not contain substr.

substringAfterLast : `string substringAfterLast(string s, string substr)`
Returns the substring of s after the last occurrence of substr, or the empty string if s does not contain substr.

substringBefore : `string substringBefore(string s, string substr)`
Returns the substring of s before the first occurrence of substr, or the empty string if s does not contain substr.

substringBeforeLast : `string substringBeforeLast(string s, string substr)`
Returns the substring of s before the last occurrence of substr, or the empty string if s does not contain substr.

tail : `string tail(string s, int len)`
Returns the last len character of s, or the full s if it is shorter than len characters.

toLower : `string toLower(string s)`
Converts s to all lowercase, and returns the result.

toUpper : `string toUpper(string s)`
Converts s to all uppercase, and returns the result.

trim : `string trim(string s)`
Discards whitespace from the start and end of s, and returns the result.

Category "units":

convertUnit : `quantity convertUnit(quantity x, string unit)`
Converts x to the given unit.

dropUnit : `double dropUnit(quantity x)`
Removes the unit of measurement from quantity x.

replaceUnit : `quantity replaceUnit(quantity x, string unit)`
Replaces the unit of x with the given unit.

unitOf : `string unitOf(quantity x)`
Returns the unit of the given quantity.

Category "xml":

xml : `xml xml(string xmlstring, string xpath?)`
Parses the given XML string into a cXMLElement tree, and returns the root element. When called with two arguments, it returns the first element from the tree that matches the expression given in simplified XPath syntax.

xmldoc : `xml xmldoc(string filename, string xpath?)`
Parses the given XML file into a cXMLElement tree, and returns the root element. When called with two arguments, it returns the first element from the tree that matches the expression given in simplified XPath syntax.

Appendix E

Message Definitions Grammar

This appendix contains the grammar for the message definitions language.

In the language, space, horizontal tab and new line characters count as delimiters, so one or more of them is required between two elements of the description which would otherwise be unseparable.

'//' (two slashes) may be used to write comments that last to the end of the line.

The language is fully case sensitive.

Notation:

- rule syntax is that of *bison/yacc*
- uppercase words are terminals, lowercase words are nonterminals
- `NAME`, `CHARCONSTANT`, `STRINGCONSTANT`, `INTCONSTANT`, `REALCONSTANT` represent identifier names and string, character, integer and real number literals (defined as in the C language)
- other terminals represent keywords in all lowercase

Nonterminals ending in `_old` are present so that message files from the previous versions of OMNeT++ (3.x) can be parsed.

```
msgfile
        : definitions
        ;

definitions
        : definitions definition
        |
        ;

definition
        : namespace_decl
        | cplusplus
        | struct_decl
        | class_decl
        | message_decl
```

```
        | packet_decl
        | enum_decl
        | enum
        | message
        | packet
        | class
        | struct
        ;

namespace_decl
        : NAMESPACE qname0 ';'

qname0
        : qname0 DOUBLECOLON NAME
        | NAME
        ;

qname
        : DOUBLECOLON qname0
        | qname0
        ;

cplusplus
        : CPLUSPLUS '{{' ... '}}' opt_semicolon
        ;

struct_decl
        : STRUCT qname ';'
        ;

class_decl
        : CLASS qname ';'
        | CLASS NONCOBJECT qname ';'
        | CLASS qname EXTENDS qname ';'
        ;

message_decl
        : MESSAGE qname ';'
        ;

packet_decl
        : PACKET qname ';'
        ;

enum_decl
        : ENUM qname ';'
        ;

enum
        : ENUM NAME '{'
          opt_enumfields '}' opt_semicolon
        ;
```

```
opt_enumfields
        : enumfields
        |
        ;

enumfields
        : enumfields enumfield
        | enumfield
        ;

enumfield
        : NAME ';'
        | NAME '=' enumvalue ';'
        ;

message
        : message_header body
        ;

packet
        : packet_header body
        ;

class
        : class_header body
        ;

struct
        : struct_header body
        ;

message_header
        : MESSAGE NAME '{'
        | MESSAGE NAME EXTENDS qname '{'
        ;

packet_header
        : PACKET NAME '{'
        | PACKET NAME EXTENDS qname '{'
        ;

class_header
        : CLASS NAME '{'
        | CLASS NAME EXTENDS qname '{'
        ;

struct_header
        : STRUCT NAME '{'
        | STRUCT NAME EXTENDS qname '{'
        ;
```

```
body
        : opt_fields_and_properties
          opt_propertiesblock_old
          opt_fieldsblock_old
          '}' opt_semicolon
        ;

opt_fields_and_properties
        : fields_and_properties
        |
        ;

fields_and_properties
        : fields_and_properties field
        | fields_and_properties property
        | field
        | property
        ;

field
        :  fieldtypename opt_fieldvector opt_inline_properties ';'
        |  fieldtypename opt_fieldvector opt_inline_properties '=' fieldvalue opt_inline
        ;

fieldtypename
        : fieldmodifiers fielddatatype NAME
        | fieldmodifiers NAME
        ;

fieldmodifiers
        : ABSTRACT
        | READONLY
        | ABSTRACT READONLY
        | READONLY ABSTRACT
        |
        ;

fielddatatype
        : qname
        | qname '*'
        | CHAR
        | SHORT
        | INT
        | LONG
        | UNSIGNED CHAR
        | UNSIGNED SHORT
        | UNSIGNED INT
        | UNSIGNED LONG
        | DOUBLE
        | STRING
        | BOOL
        ;
```

```
opt_fieldvector
        : '[' INTCONSTANT ']'
        | '[' qname ']'
        | '[' ']'
        |
        ;

fieldvalue
        : fieldvalue fieldvalueitem
        | fieldvalueitem
        ;

fieldvalueitem
        : STRINGCONSTANT
        | CHARCONSTANT
        | INTCONSTANT
        | REALCONSTANT
        | TRUE
        | FALSE
        | NAME
        | DOUBLECOLON
        | '?' | ':' | '&&' | '||' | '##' | '==' | '!=' | '>' | '>=' | '<' | '<='
        | '&' | '|' | '#' | '<<' | '>>'
        | '+' | '-' | '*' | '/' | '%' | '^' | '&' | UMIN | '!' | '~'
        | '.' | ',' | '(' | ')' | '[' | ']'
        ;

enumvalue
        : INTCONSTANT
        | '-' INTCONSTANT
        | NAME
        ;

opt_inline_properties
        : inline_properties
        |
        ;

inline_properties
        : inline_properties property_namevalue
        | property_namevalue
        ;

property
        : property_namevalue ';'
        ;

property_namevalue
        : property_name
        | property_name '(' opt_property_keys ')'
        | ENUM '(' NAME ')'
```

```
        ;

property_name
        : '@' PROPNAME
        | '@' PROPNAME '[' PROPNAME ']'
        ;

opt_property_keys
        : property_keys
        ;

property_keys
        : property_keys ';' property_key
        | property_key
        ;

property_key
        : property_literal '=' property_values
        | property_values
        ;

property_values
        : property_values ',' property_value
        | property_value
        ;

property_value
        : property_literal
        |
        ;

property_literal
        : property_literal CHAR
        | property_literal STRINGCONSTANT
        | CHAR
        | STRINGCONSTANT
        ;

opt_fieldsblock_old
        : FIELDS ':'
          opt_fields_old
        |
        ;

opt_fields_old
        : fields_old
        |
        ;

fields_old
        : fields_old field
        | field
```

```
        ;

opt_propertiesblock_old
        : PROPERTIES ':'
          opt_properties_old
        |
        ;

opt_properties_old
        : properties_old
        |
        ;

properties_old
        : properties_old property_old
        | property_old
        ;

property_old
        : NAME '=' property_value ';'
        ;

opt_semicolon : ';' | ;
```

Appendix F

Display String Tags

F.1 Module and Connection Display String Tags

Supported module and connection display string tags are listed in the following table.

Tag[argument index] - name	Description
p[0] - x	X position of the center of the icon/shape; defaults to automatic graph layouting
p[1] - y	Y position of the center of the icon/shape; defaults to automatic graph layouting
p[2] - arrangement	Arrangement of submodule vectors. Values: row (r), column (c), matrix (m), ring (ri), exact (x)
p[3] - arr. par1	Depends on arrangement: matrix => ncols, ring => rx, exact => dx, row => dx, column => dy
p[4] - arr. par2	Depends on arrangement: matrix => dx, ring => ry, exact => dy
p[5] - arr. par3	Depends on arrangement: matrix => dy
b[0] - width	Width of object. Default: 40
b[1] - height	Height of object. Default: 24
b[2] - shape	Shape of object. Values: rectangle (rect), oval (oval). Default: rect
b[3] - fill color	Fill color of the object (colorname or #RRGGBB or @HHSSBB). Default: #8080ff
b[4] - border color	Border color of the object (colorname or #RRGGBB or @HHSSBB). Default: black
b[5] - border width	Border width of the object. Default: 2
i[0] - icon	An icon representing the object
i[1] - icon color	A color to colorize the icon (colorname or #RRGGBB or @HHSSBB)
i[2] - icon colorization	Amount of colorization in percent. Default: 30
is[0] - icon size	The size of the image. Values: very small (vs), small (s), normal (n), large (l), very large (vl)
i2[0] - overlay icon	An icon added to the upper right corner of the original image

i2[1] - overlay icon color	A color to colorize the overlay icon (colorname or #RRGGBB or @HHSSBB)
i2[2] - overlay icon colorization	Amount of colorization in percent. Default: 30
r[0] - range	Radius of the range indicator
r[1] - range fill color	Fill color of the range indicator (colorname or #RRGGBB or @HHSSBB)
r[2] - range border color	Border color of the range indicator (colorname or #RRGGBB or @HHSSBB). Default: black
r[3] - range border width	Border width of the range indicator. Default: 1
q[0] - queue object	Displays the length of the named queue object
t[0] - text	Additional text to display
t[1] - text position	Position of the text. Values: left (l), right (r), top (t). Default: t
t[2] - text color	Color of the displayed text (colorname or #RRGGBB or @HHSSBB). Default: blue
tt[0] - tooltip	Tooltip to be displayed over the object
bgp[0] - bg x	Module background horizontal offset
bgp[1] - bg y	Module background vertical offset
bgb[0] - bg width	Width of the module background rectangle
bgb[1] - bg height	Height of the module background rectangle
bgb[2] - bg fill color	Background fill color (colorname or #RRGGBB or @HHSSBB). Default: grey82
bgb[3] - bg border color	Border color of the module background rectangle (colorname or #RRGGBB or @HHSSBB). Default: black
bgb[4] - bg border width	Border width of the module background rectangle. Default: 2
bgtt[0] - bg tooltip	Tooltip to be displayed over the module's background
bgi[0] - bg image	An image to be displayed as a module background
bgi[1] - bg image mode	How to arrange the module's background image. Values: fix (f), tile (t), stretch (s), center (c). Default: fixed
bgg[0] - grid tick distance	Distance between two major ticks measured in units
bgg[1] - grid minor ticks	Minor ticks per major ticks. Default: 1
bgg[2] - grid color	Color of the grid lines (colorname or #RRGGBB or @HHSSBB). Default: grey
bgl[0] - layout seed	Seed value for layout algorithm
bgl[1] - layout algorithm	Algorithm for child layouting
bgs[0] - pixels per unit	Number of pixels per distance unit
bgs[1] - unit name	Name of distance unit
m[0] - routing constraint	Connection routing constraint. Values: auto (a), south (s), north (n), east (e), west (w)
ls[0] - line color	Connection color (colorname or #RRGGBB or @HHSSBB). Default: black
ls[1] - line width	Connection line width. Default: 1
ls[2] - line style	Connection line style. Values: solid (s), dotted (d), dashed (da). Default: solid

F.2 Message Display String Tags

To customize the appearance of messages in the graphical runtime environment, override the `getDisplayString()` method of `cMessage` or `cPacket` to return a display string.

Tag	Meaning
b=*width*,*height*,**oval**	Ellipse with the given *height* and *width*. Defaults: *width*=10, *height*=10
b=*width*,*height*,**rect**	Rectangle with the given *height* and *width*. Defaults: *width*=10, *height*=10
o=*fillcolor*,*outlinecolor*,*borderwidth*	Specifies options for the rectangle or oval. For color notation, see section 8.3. Defaults: *fillcolor*=red, *outlinecolor*=black, *borderwidth*=1
i=*iconname*,*color*,*percentage*	Use the named icon. It can be colorized, and percentage specifies the amount of colorization. If color name is `"kind"`, a message kind dependent colors is used (like default behaviour). Defaults: *iconname*: no default – if no icon name is present, a small red solid circle will be used; *color*: no coloring; *percentage*: 30%
tt=*tooltip-text*	Displays the given text in a tooltip when the user moves the mouse over the message icon.

Appendix G

Configuration Options

G.1 Configuration Options

This section lists all configuration options that are available in ini files. A similar list can be obtained from any simulation executable by running it with the `-h configdetails` option.

```
check-signals=<bool>, default:false; per-run setting
    Controls whether the simulation kernel will validate signals emitted by
    modules and channels against signal declarations (@signal properties) in
    NED files.

cmdenv-autoflush=<bool>, default:false; per-run setting
    Call fflush(stdout) after each event banner or status update; affects both
    express and normal mode. Turning on autoflush may have a performance
    penalty, but it can be useful with printf-style debugging for tracking down
    program crashes.

cmdenv-config-name=<string>; global setting
    Specifies the name of the configuration to be run (for a value 'Foo',
    section [Config Foo] will be used from the ini file). See also
    cmdenv-runs-to-execute=. The -c command line option overrides this setting.

<object-full-path>.cmdenv-ev-output=<bool>, default:true; per-object setting
    When cmdenv-express-mode=false: whether Cmdenv should print debug messages
    (ev<<) from the selected modules.

cmdenv-event-banner-details=<bool>, default:false; per-run setting
    When cmdenv-express-mode=false: print extra information after event
    banners.

cmdenv-event-banners=<bool>, default:true; per-run setting
    When cmdenv-express-mode=false: turns printing event banners on/off.

cmdenv-express-mode=<bool>, default:true; per-run setting
    Selects ''normal'' (debug/trace) or ''express'' mode.
```

```
cmdenv-extra-stack=<double>, unit="B", default:8KiB; global setting
    Specifies the extra amount of stack that is reserved for each activity()
    simple module when the simulation is run under Cmdenv.

cmdenv-interactive=<bool>, default:false; global setting
    Defines what Cmdenv should do when the model contains unassigned
    parameters. In interactive mode, it asks the user. In non-interactive mode
    (which is more suitable for batch execution), Cmdenv stops with an error.

cmdenv-message-trace=<bool>, default:false; per-run setting
    When cmdenv-express-mode=false: print a line per message sending (by
    send(),scheduleAt(), etc) and delivery on the standard output.

cmdenv-module-messages=<bool>, default:true; per-run setting
    When cmdenv-express-mode=false: turns printing module ev<< output on/off.

cmdenv-output-file=<filename>; global setting
    When a filename is specified, Cmdenv redirects standard output into the
    given file. This is especially useful with parallel simulation. See the
    'fname-append-host' option as well.

cmdenv-performance-display=<bool>, default:true; per-run setting
    When cmdenv-express-mode=true: print detailed performance information.
    Turning it on results in a 3-line entry printed on each update, containing
    ev/sec, simsec/sec, ev/simsec, number of messages created/still
    present/currently scheduled in FES.

cmdenv-runs-to-execute=<string>; global setting
    Specifies which runs to execute from the selected configuration (see
    cmdenv-config-name=). It accepts a comma-separated list of run numbers or
    run number ranges, e.g. 1,3..4,7..9. If the value is missing, Cmdenv
    executes all runs in the selected configuration. The -r command line option
    overrides this setting.

cmdenv-status-frequency=<double>, unit="s", default:2s; per-run setting
    When cmdenv-express-mode=true: print status update every n seconds.

configuration-class=<string>; global setting
    Part of the Envir plugin mechanism: selects the class from which all
    configuration information will be obtained. This option lets you replace
    omnetpp.ini with some other implementation, e.g. database input. The
    simulation program still has to bootstrap from an omnetpp.ini (which
    contains the configuration-class setting). The class should implement the
    cConfigurationEx interface.

constraint=<string>; per-run setting
    For scenarios. Contains an expression that iteration variables (${} syntax)
    must satisfy for that simulation to run. Example: $i < $j+1.

cpu-time-limit=<double>, unit="s"; per-run setting
    Stops the simulation when CPU usage has reached the given limit. The
    default is no limit.
```

```
debug-on-errors=<bool>, default:false; global setting
    When set to true, runtime errors will cause the simulation program to break
    into the C++ debugger (if the simulation is running under one, or
    just-in-time debugging is activated). Once in the debugger, you can view
    the stack trace or examine variables.

debug-statistics-recording=<bool>, default:false; per-run setting
    Turns on the printing of debugging information related to statistics
    recording (@statistic properties)

debugger-attach-command=<string>, default:nemiver --attach=%u &; global setting
    Command line to launch the debugger. It must contain exactly one percent
    sign, as '%u', which will be replaced by the PID of this process. The
    command must not block (i.e. it should end in '&' on Unix-like systems).

debugger-attach-on-error=<bool>, default:false; global setting
    When set to true, runtime errors and crashes will trigger an external
    debugger to be launched, allowing you to do just-in-time debugging on the
    simulation process.

debugger-attach-on-startup=<bool>, default:false; global setting
    When set to true, the simulation program will launch an external debugger
    attached to it, allowing you to set breakpoints before proceeding. The
    debugger command is configurable.

debugger-attach-wait-time=<double>, unit="s", default:20s; global setting
    An interval to wait after launching the external debugger, to give the
    debugger time to start up and attach to the simulation process.

description=<string>; per-run setting
    Descriptive name for the given simulation configuration. Descriptions get
    displayed in the run selection dialog.

eventlog-file=<filename>, default:${resultdir}/${configname}-${runnumber}.elog; per-run
    Name of the eventlog file to generate.

eventlog-message-detail-pattern=<custom>; per-run setting
    A list of patterns separated by '|' character which will be used to write
    message detail information into the eventlog for each message sent during
    the simulation. The message detail will be presented in the sequence chart
    tool. Each pattern starts with an object pattern optionally followed by ':'
    character and a comma separated list of field patterns. In both patterns
    and/or/not/* and various field match expressions can be used. The object
    pattern matches to class name, the field pattern matches to field name by
    default.
      EVENTLOG-MESSAGE-DETAIL-PATTERN := ( DETAIL-PATTERN '|' )* DETAIL_PATTERN
      DETAIL-PATTERN := OBJECT-PATTERN [ ':' FIELD-PATTERNS ]
      OBJECT-PATTERN := MATCH-EXPRESSION
      FIELD-PATTERNS := ( FIELD-PATTERN ',' )* FIELD_PATTERN
      FIELD-PATTERN := MATCH-EXPRESSION
    Examples (enter them without quotes):
```

```
      "*": captures all fields of all messages
      "*Frame:*Address,*Id": captures all fields named somethingAddress and
    somethingId from messages of any class named somethingFrame
      "MyMessage:declaredOn(MyMessage)": captures instances of MyMessage
    recording the fields declared on the MyMessage class
      "*:(not declaredOn(cMessage) and not declaredOn(cNamedObject) and not
    declaredOn(cObject))": records user-defined fields from all messages

eventlog-recording-intervals=<custom>; per-run setting
    Simulation time interval(s) when events should be recorded. Syntax:
    [<from>]..[<to>],... That is, both start and end of an interval are
    optional, and intervals are separated by comma. Example: ..10.2, 22.2..100,
    233.3..

experiment-label=<string>, default:${configname}; per-run setting
    Identifies the simulation experiment (which consists of several,
    potentially repeated measurements). This string gets recorded into result
    files, and may be referred to during result analysis.

extends=<string>; per-run setting
    Name of the configuration this section is based on. Entries from that
    section will be inherited and can be overridden. In other words,
    configuration lookups will fall back to the base section.

fingerprint=<string>; per-run setting
    The expected fingerprint of the simulation. When provided, a fingerprint
    will be calculated from the simulation event times and other quantities
    during simulation, and checked against the given one. Fingerprints are
    suitable for crude regression tests. As fingerprints occasionally differ
    across platforms, more than one fingerprint values can be specified here,
    separated by spaces, and a match with any of them will be accepted. To
    obtain the initial fingerprint, enter a dummy value such as \"0000\"),
    and run the simulation.

fname-append-host=<bool>; global setting
    Turning it on will cause the host name and process Id to be appended to the
    names of output files (e.g. omnetpp.vec, omnetpp.sca). This is especially
    useful with distributed simulation. The default value is true if parallel
    simulation is enabled, false otherwise.

load-libs=<filenames>; global setting
    A space-separated list of dynamic libraries to be loaded on startup. The
    libraries should be given without the '.dll' or '.so' suffix -- that will
    be automatically appended.

max-module-nesting=<int>, default:50; per-run setting
    The maximum allowed depth of submodule nesting. This is used to catch
    accidental infinite recursions in NED.

measurement-label=<string>, default:${iterationvars}; per-run setting
    Identifies the measurement within the experiment. This string gets recorded
    into result files, and may be referred to during result analysis.
```

```
<object-full-path>.module-eventlog-recording=<bool>, default:true; per-object setting
    Enables recording events on a per module basis. This is meaningful for
    simple modules only.
    Example:
     **.router[10..20].**.module-eventlog-recording = true
     **.module-eventlog-recording = false

ned-path=<path>; global setting
    A semicolon-separated list of directories. The directories will be regarded
    as roots of the NED package hierarchy, and all NED files will be loaded
    from their subdirectory trees. This option is normally left empty, as the
    OMNeT++ IDE sets the NED path automatically, and for simulations started
    outside the IDE it is more convenient to specify it via a command-line
    option or the NEDPATH environment variable.

network=<string>; per-run setting
    The name of the network to be simulated.  The package name can be omitted
    if the ini file is in the same directory as the NED file that contains the
    network.

num-rngs=<int>, default:1; per-run setting
    The number of random number generators.

output-scalar-file=<filename>, default:${resultdir}/${configname}-${runnumber}.sca; per-
    Name for the output scalar file.

output-scalar-file-append=<bool>, default:false; per-run setting
    What to do when the output scalar file already exists: append to it
    (OMNeT++ 3.x behavior), or delete it and begin a new file (default).

output-scalar-precision=<int>, default:14; per-run setting
    The number of significant digits for recording data into the output scalar
    file. The maximum value is ~15 (IEEE double precision).

output-vector-file=<filename>, default:${resultdir}/${configname}-${runnumber}.vec; per-
    Name for the output vector file.

output-vector-precision=<int>, default:14; per-run setting
    The number of significant digits for recording data into the output vector
    file. The maximum value is ~15 (IEEE double precision). This setting has no
    effect on the "time" column of output vectors, which are represented as
    fixed-point numbers and always get recorded precisely.

output-vectors-memory-limit=<double>, unit="B", default:16MiB; per-run setting
    Total memory that can be used for buffering output vectors. Larger values
    produce less fragmented vector files (i.e. cause vector data to be grouped
    into larger chunks), and therefore allow more efficient processing later.

outputscalarmanager-class=<string>, default:cFileOutputScalarManager; global setting
    Part of the Envir plugin mechanism: selects the output scalar manager class
    to be used to record data passed to recordScalar(). The class has to
```

```
    implement the cOutputScalarManager interface.

outputvectormanager-class=<string>, default:cIndexedFileOutputVectorManager; global setti
    Part of the Envir plugin mechanism: selects the output vector manager class
    to be used to record data from output vectors. The class has to implement
    the cOutputVectorManager interface.

parallel-simulation=<bool>, default:false; global setting
    Enables parallel distributed simulation.

<object-full-path>.param-record-as-scalar=<bool>, default:false; per-object setting
    Applicable to module parameters: specifies whether the module parameter
    should be recorded into the output scalar file. Set it for parameters whose
    value you'll need for result analysis.

parsim-communications-class=<string>, default:cFileCommunications; global setting
    If parallel-simulation=true, it selects the class that implements
    communication between partitions. The class must implement the
    cParsimCommunications interface.

parsim-debug=<bool>, default:true; global setting
    With parallel-simulation=true: turns on printing of log messages from the
    parallel simulation code.

parsim-filecommunications-prefix=<string>, default:comm/; global setting
    When cFileCommunications is selected as parsim communications class:
    specifies the prefix (directory+potential filename prefix) for creating the
    files for cross-partition messages.

parsim-filecommunications-preserve-read=<bool>, default:false; global setting
    When cFileCommunications is selected as parsim communications class:
    specifies that consumed files should be moved into another directory
    instead of being deleted.

parsim-filecommunications-read-prefix=<string>, default:comm/read/; global setting
    When cFileCommunications is selected as parsim communications class:
    specifies the prefix (directory) where files will be moved after having
    been consumed.

parsim-idealsimulationprotocol-tablesize=<int>, default:100000; global setting
    When cIdealSimulationProtocol is selected as parsim synchronization class:
    specifies the memory buffer size for reading the ISP event trace file.

parsim-mpicommunications-mpibuffer=<int>; global setting
    When cMPICommunications is selected as parsim communications class:
    specifies the size of the MPI communications buffer. The default is to
    calculate a buffer size based on the number of partitions.

parsim-namedpipecommunications-prefix=<string>, default:comm/; global setting
    When cNamedPipeCommunications is selected as parsim communications class:
    selects the prefix (directory+potential filename prefix) where name pipes
    are created in the file system.
```

```
parsim-nullmessageprotocol-laziness=<double>, default:0.5; global setting
    When cNullMessageProtocol is selected as parsim synchronization class:
    specifies the laziness of sending null messages. Values in the range [0,1)
    are accepted. Laziness=0 causes null messages to be sent out immediately as
    a new EOT is learned, which may result in excessive null message traffic.

parsim-nullmessageprotocol-lookahead-class=<string>, default:cLinkDelayLookahead; global
    When cNullMessageProtocol is selected as parsim synchronization class:
    specifies the C++ class that calculates lookahead. The class should
    subclass from cNMPLookahead.

parsim-synchronization-class=<string>, default:cNullMessageProtocol; global setting
    If parallel-simulation=true, it selects the parallel simulation algorithm.
    The class must implement the cParsimSynchronizer interface.

<object-full-path>.partition-id=<string>; per-object setting
    With parallel simulation: in which partition the module should be
    instantiated. Specify numeric partition ID, or a comma-separated list of
    partition IDs for compound modules that span across multiple partitions.
    Ranges ("5..9") and "*" (=all) are accepted too.

print-undisposed=<bool>, default:true; global setting
    Whether to report objects left (that is, not deallocated by simple module
    destructors) after network cleanup.

realtimescheduler-scaling=<double>; global setting
    When cRealTimeScheduler is selected as scheduler class: ratio of simulation
    time to real time. For example, scaling=2 will cause simulation time to
    progress twice as fast as runtime.

record-eventlog=<bool>, default:false; per-run setting
    Enables recording an eventlog file, which can be later visualized on a
    sequence chart. See eventlog-file= option too.

repeat=<int>, default:1; per-run setting
    For scenarios. Specifies how many replications should be done with the same
    parameters (iteration variables). This is typically used to perform
    multiple runs with different random number seeds. The loop variable is
    available as ${repetition}. See also: seed-set= key.

replication-label=<string>, default:#${repetition}; per-run setting
    Identifies one replication of a measurement (see repeat= and
    measurement-label= as well). This string gets recorded into result files,
    and may be referred to during result analysis.

result-dir=<string>, default:results; per-run setting
    Value for the ${resultdir} variable, which is used as the default directory
    for result files (output vector file, output scalar file, eventlog file,
    etc.)

<object-full-path>.result-recording-modes=<string>, default:default; per-object setting
```

```
    Defines how to calculate results from the @statistic property matched by
    the wildcard. Special values: default, all: they select the modes listed in
    the record= key of @statistic; all selects all of them, default selects the
    non-optional ones (i.e. excludes the ones that end in a question mark).
    Example values: vector, count, last, sum, mean, min, max, timeavg, stats,
    histogram. More than one values are accepted, separated by commas.
    Expressions are allowed. Items prefixed with '-' get removed from the list.
    Example: **.queueLength.result-recording-modes=default,-vector,+timeavg

<object-full-path>.rng-%=<int>; per-object setting
    Maps a module-local RNG to one of the global RNGs. Example: **.gen.rng-1=3
    maps the local RNG 1 of modules matching '**.gen' to the global RNG 3. The
    default is one-to-one mapping.

rng-class=<string>, default:cMersenneTwister; per-run setting
    The random number generator class to be used. It can be 'cMersenneTwister',
    'cLCG32', 'cAkaroaRNG', or you can use your own RNG class (it must be
    subclassed from cRNG).

runnumber-width=<int>, default:0; per-run setting
    Setting a nonzero value will cause the $runnumber variable to get padded
    with leading zeroes to the given length.

<object-full-path>.scalar-recording=<bool>, default:true; per-object setting
    Whether the matching output scalars should be recorded. Syntax:
    <module-full-path>.<scalar-name>.scalar-recording=true/false. Example:
    **.queue.packetsDropped.scalar-recording=true

scheduler-class=<string>, default:cSequentialScheduler; global setting
    Part of the Envir plugin mechanism: selects the scheduler class. This
    plugin interface allows for implementing real-time, hardware-in-the-loop,
    distributed and distributed parallel simulation. The class has to implement
    the cScheduler interface.

sectionbasedconfig-configreader-class=<string>; global setting
    When configuration-class=SectionBasedConfiguration: selects the
    configuration reader C++ class, which must subclass from
    cConfigurationReader.

seed-%-lcg32=<int>; per-run setting
    When cLCG32 is selected as random number generator: seed for the kth RNG.
    (Substitute k for '%' in the key.)

seed-%-mt=<int>; per-run setting
    When Mersenne Twister is selected as random number generator (default):
    seed for RNG number k. (Substitute k for '%' in the key.)

seed-%-mt-p%=<int>; per-run setting
    With parallel simulation: When Mersenne Twister is selected as random
    number generator (default): seed for RNG number k in partition number p.
    (Substitute k for the first '%' in the key, and p for the second.)
```

```
seed-set=<int>, default:${runnumber}; per-run setting
    Selects the kth set of automatic random number seeds for the simulation.
    Meaningful values include ${repetition} which is the repeat loop counter
    (see repeat= key), and ${runnumber}.

sim-time-limit=<double>, unit="s"; per-run setting
    Stops the simulation when simulation time reaches the given limit. The
    default is no limit.

simtime-scale=<int>, default:-12; global setting
    Sets the scale exponent, and thus the resolution of time for the 64-bit
    fixed-point simulation time representation. Accepted values are -18..0; for
    example, -6 selects microsecond resolution. -12 means picosecond
    resolution, with a maximum simtime of ~110 days.

snapshot-file=<filename>, default:${resultdir}/${configname}-${runnumber}.sna; per-run s
    Name of the snapshot file.

snapshotmanager-class=<string>, default:cFileSnapshotManager; global setting
    Part of the Envir plugin mechanism: selects the class to handle streams to
    which snapshot() writes its output. The class has to implement the
    cSnapshotManager interface.

tkenv-default-config=<string>; global setting
    Specifies which config Tkenv should set up automatically on startup. The
    default is to ask the user.

tkenv-default-run=<int>, default:0; global setting
    Specifies which run (of the default config, see tkenv-default-config) Tkenv
    should set up automatically on startup. The default is to ask the user.

tkenv-extra-stack=<double>, unit="B", default:48KiB; global setting
    Specifies the extra amount of stack that is reserved for each activity()
    simple module when the simulation is run under Tkenv.

tkenv-image-path=<path>; global setting
    Specifies the path for loading module icons.

tkenv-plugin-path=<path>; global setting
    Specifies the search path for Tkenv plugins. Tkenv plugins are .tcl files
    that get evaluated on startup.

total-stack=<double>, unit="B"; global setting
    Specifies the maximum memory for activity() simple module stacks. You need
    to increase this value if you get a ``Cannot allocate coroutine stack''
    error.

<object-full-path>.typename=<string>; per-object setting
    Specifies type for submodules and channels declared with 'like <>'.

user-interface=<string>; global setting
    Selects the user interface to be started. Possible values are Cmdenv and
```

```
    Tkenv. This option is normally left empty, as it is more convenient to
    specify the user interface via a command-line option or the IDE's Run and
    Debug dialogs. New user interfaces can be defined by subclassing
    cRunnableEnvir.

<object-full-path>.vector-max-buffered-values=<int>; per-object setting
    For output vectors: the maximum number of values to buffer per vector,
    before writing out a block into the output vector file. The default is no
    per-vector limit (i.e. only the total memory limit is in effect)

<object-full-path>.vector-record-eventnumbers=<bool>, default:true; per-object setting
    Whether to record event numbers for an output vector. Simulation time and
    value are always recorded. Event numbers are needed by the Sequence Chart
    Tool, for example.

<object-full-path>.vector-recording=<bool>, default:true; per-object setting
    Whether data written into an output vector should be recorded.

<object-full-path>.vector-recording-intervals=<custom>; per-object setting
    Recording interval(s) for an output vector. Syntax: [<from>]..[<to>],...
    That is, both start and end of an interval are optional, and intervals are
    separated by comma. Example: ..100, 200..400, 900..

warmup-period=<double>, unit="s"; per-run setting
    Length of the initial warm-up period. When set, results belonging to the
    first x seconds of the simulation will not be recorded into output vectors,
    and will not be counted into output scalars (see option
    **.result-recording-modes). This option is useful for steady-state
    simulations. The default is 0s (no warmup period). Note that models that
    compute and record scalar results manually (via recordScalar()) will not
    automatically obey this setting.

warnings=<bool>, default:true; per-run setting
    Enables warnings.

<object-full-path>.with-akaroa=<bool>, default:false; per-object setting
    Whether to the output vector should be under Akaroa control.
```

G.2 Predefined Configuration Variables

Predefined variables that can be used in config values:

```
${runid}
    A reasonably globally unique identifier for the run, produced by
    concatenating the configuration name, run number, date/time, etc.
${inifile}
    Name of the (primary) inifile
${configname}
    Name of the active configuration
${runnumber}
```

```
    Sequence number of the current run within all runs in the active
    configuration
${network}
    Value of the "network" configuration option
${experiment}
    Value of the "experiment-label" configuration option
${measurement}
    Value of the "measurement-label" configuration option
${replication}
    Value of the "replication-label" configuration option
${processid}
    PID of the simulation process
${datetime}
    Date and time the simulation run was started
${resultdir}
    Value of the "result-dir" configuration option
${repetition}
    The iteration number in 0..N-1, where N is the value of the "repeat"
    configuration option
${seedset}
    Value of the "seed-set" configuration option
${iterationvars}
    Concatenation of all user-defined iteration variables in name=value form
${iterationvars2}
    Concatenation of all user-defined iteration variables in name=value form,
    plus ${repetition}
```

Appendix H

Result File Formats

The file format described here applies to *both output vector and output scalar files*. Their formats are consistent, only the types of entries occurring in them are different. This unified format also means that they can be read with a common routine.

Result files are *line oriented*. A line consists of one or more tokens, separated by whitespace. Tokens either do not contain whitespace, or whitespace is escaped using a backslash, or are quoted using double quotes. Escaping within quotes using backslashes is also permitted.

The first token of a line usually identifies the type of the entry. A notable exception is an output vector data line, which begins with a numeric identifier of the given output vector.

A line starting with # as the first non-whitespace character denotes a comment, and is to be ignored during processing.

Result files are written from simulation runs. A simulation run generates physically contiguous sets of lines into one or more result files. (That is, lines from different runs do not arbitrarily mix in the files.)

A run is identified by a unique textual *runId*, which appears in all result files written during that run. The runId may appear on the user interface, so it should be somewhat meaningful to the user. Nothing should be assumed about the particular format of runId, but it will be some string concatenated from the simulated network's name, the time/date, the hostname, and other pieces of data to make it unique.

A simulation run will typically write into two result files (.vec and .sca). However, when using parallel distributed simulation, the user will end up with several .vec and .sca files, because different partitions (a separate process each) will write into different files. However, all these files will contain the same runId, so it is possible to relate data that belong together.

Entry types are:

- **version**: result file version
- **run**: simulation run identifier
- **attr**: run, vector, scalar or statistics object attribute
- **param**: module parameter
- **scalar**: scalar data
- **vector**: vector declaration

- *vector-id*: vector data
- **file**: vector file attributes
- **statistic**: statistics object
- **field**: field of a statistics object
- **bin**: histogram bin

H.1 Version

Specifies the format of the result file. It is written at the beginning of the file.

Syntax:

version *versionNumber*

The version described in this document is 2. Version 1 files are produced by OMNeT++ 3.3 or earlier.

H.2 Run Declaration

Marks the beginning of a new run in the file. Entries after this line belong to this run.

Syntax:

run *runId*

Example:

```
run TokenRing1-0-20080514-18:19:44-3248
```

Typically there will be one run per file, but this is not mandatory. In cases when there are more than one run in a file and it is not feasible to keep the entire file in memory during analysis, the offsets of the *run* lines may be indexed for more efficient random access.

The *run* line may be immediately followed by *attribute* lines. Attributes may store generic data like the network name, date/time of running the simulation, configuration options that took effect for the simulation, etc.

Run attribute names used by OMNeT++ include the following:

Generic attribute names:

- **network**: name of the network simulated
- **datetime**: date/time associated with the run
- **processid**: the PID of the simulation process
- **inifile**: the main configuration file
- **configname**: name of the inifile configuration
- **seedset**: index of the seed-set use for the simulation

Attributes associated with parameter studies (iterated runs):

- **runnumber**: the run number within the parameter study
- **experiment**: experiment label
- **measurement**: measurement label
- **replication**: replication label
- **repetition**: the loop counter for repetitions with different seeds
- **iterationvars**: string containing the values of the iteration variables
- **iterationvars2**: string containing the values of the iteration variables

An example run header:

```
run TokenRing1-0-20080514-18:19:44-3248
attr configname TokenRing1
attr datetime 20080514-18:19:44
attr experiment TokenRing1
attr inifile omnetpp.ini
attr iterationvars ""
attr iterationvars2 $repetition=0
attr measurement ""
attr network TokenRing
attr processid 3248
attr repetition 0
attr replication #0
attr resultdir results
attr runnumber 0
attr seedset 0
```

H.3 Attributes

Contains an attribute for the preceding run, vector, scalar or statistics object. Attributes can be used for saving arbitrary extra information for objects; processors should ignore unrecognized attributes.

Syntax:

attr *name value*

Example:

```
attr network "largeNet"
```

H.4 Module Parameters

Contains a module parameter value for the given run. This is needed so that module parameters may be included in the analysis (e.g. to identify the load for a "throughput vs load" plot).

It may not be practical to simply store all parameters of all modules in the result file, because there may be too many. We assume that NED files are invariant and do not store parameters

defined in them. However, we store parameter assignments that come from `omnetpp.ini`, in their original wildcard form (i.e. not expanded) to conserve space. Parameter values entered interactively by the user are also stored.

When the original NED files are present, it should thus be possible to reconstruct all parameters for the given simulation.

Syntax:

param *parameterNamePattern value*

Example:

```
param **.gen.sendIaTime  exponential(0.01)
param **.gen.msgLength   10
param **.fifo.bitsPerSec 1000
```

H.5 Scalar Data

Contains an output scalar value.

Syntax:

scalar *moduleName scalarName value*

Examples:

```
scalar "net.switchA.relay" "processed frames" 100
```

Scalar lines may be immediately followed by *attribute* lines. OMNeT++ uses the following attributes for scalars:

- **title**: suggested title on charts
- **unit**: measurement unit, e.g. `s` for seconds

H.6 Vector Declaration

Defines an output vector.

Syntax:

vector *vectorId moduleName vectorName*

vector *vectorId moduleName vectorName columnSpec*

Where *columnSpec* is a string, encoding the meaning and ordering the columns of data lines. Characters of the string mean:

- **E** event number
- **T** simulation time
- **V** vector value

Common values are `TV` and `ETV`. The default value is `TV`.

Vector lines may be immediately followed by *attribute* lines. OMNeT++ uses the following attributes for vectors:

- **title**: suggested vector title on charts
- **unit**: measurement unit, e.g. `s` for seconds
- **enum**: symbolic names for values of the vector; syntax is `"IDLE=0, BUSY=1, OFF=2"`
- **type**: data type, one of `int`, `double` and `enum`
- **interpolationmode**: hint for interpolation mode on the chart: `none` (=do not connect the dots), `sample-hold`, `backward-sample-hold`, `linear`
- **min**: minimum value
- **max**: maximum value

H.7 Vector Data

Adds a value to an output vector. This is the same as in older output vector files.

Syntax:

vectorId column1 column2 ...

Simulation times and event numbers *within an output vector* are required to be in increasing order.

Performance note: Data lines belonging to the same output vector may be written out in clusters (of size roughly a multiple of the disk's physical block size). Then, since an output vector file is typically not kept in memory during analysis, indexing the start offsets of these clusters allows one to read the file and seek in it more efficiently. This does not require any change or extension to the file format.

H.8 Index Header

The first line of the index file stores the size and modification date of the vector file. If the attributes of a vector file differ from the information stored in the index file, then the IDE automatically rebuilds the index file.

Syntax:

file *filesize modificationDate*

H.9 Index Data

Stores the location and statistics of blocks in the vector file.

Syntax:

vectorId offset length firstEventNo lastEventNo firstSimtime lastSimtime count min max sum sqrsum

where

- *offset*: the start offset of the block
- *length*: the length of the block

- *firstEventNo*, *lastEventNo*: the event number range of the block (optional)
- *firstSimtime*, *lastSimtime*: the simtime range of the block
- *count, min, max, sum, sqrsum*: collected statistics of the values in the block

H.10 Statistics Object

Represents a statistics object.

Syntax:

statistic *moduleName statisticName*

Example:

```
statistic Aloha.server  "collision multiplicity"
```

A *statistic* line may be followed by *field* and *attribute* lines, and a series of *bin* lines that represent histogram data.

OMNeT++ uses the following attributes:

- **title**: suggested title on charts
- **unit**: measurement unit, e.g. `s` for seconds
- **type**: type of the collected values: `int` or `double`; the default is `double`

A full example with fields, attributes and histogram bins:

```
statistic Aloha.server  "collision multiplicity"
field count 13908
field mean 6.8510209951107
field stddev 5.2385484477843
field sum 95284
field sqrsum 1034434
field min 2
field max 65
attr type int
attr unit packets
bin     -INF    0
bin     0       0
bin     1       0
bin     2       2254
bin     3       2047
bin     4       1586
bin     5       1428
bin     6       1101
bin     7       952
bin     8       785
...
bin     52      2
```

H.11 Field

Represents a field in a statistics object.

Syntax:

field *fieldName value*

Example:

```
field sum 95284
```

Fields:

- **count**: observation count
- **mean**: mean of the observations
- **stddev**: standard deviation
- **sum**: sum of the observations
- **sqrsum**: sum of the squared observations
- **min**: minimum of the observations
- **max**: maximum of the observations

For weighted statistics, additionally the following fields may be recorded:

- **weights**: sum of the weights
- **weightedSum**: the weighted sum of the observations
- **sqrSumWeights**: sum of the squared weights
- **weightedSqrSum**: weighted sum of the squared observations

H.12 Histogram Bin

Represents a bin in a histogram object.

Syntax:

bin *binLowerBound value*

Histogram name and module is defined on the **statistic** line, which is followed by several **bin** lines to contain data. Any non-**bin** line marks the end of the histogram data.

The *binLowerBound* column of **bin** lines represent the (inclusive) lower bound of the given histogram cell. **Bin** lines are in increasing *binLowerBound* order.

The *value* column of a **bin** line represents the observation count in the given cell: *value k* is the number of observations greater or equal to *binLowerBound k*, but smaller than *binLowerBound k+1*. *Value* is not necessarily an integer, because the cKSplit and cPSquare algorithms produce non-integer estimates. The first **bin** line is the underflow cell, and the last **bin** line is the overflow cell.

Example:

```
bin -INF  0
bin 0 4
bin 2 6
bin 4 2
bin 6 1
```

Appendix I

Eventlog File Format

This appendix documents the format of the eventlog file. Eventlog files are written by the simulation (when enabled). Everything that happens during the simulation is recorded into the file, [1] so the file can later be used to reproduce the history of the simulation on a sequence chart, or in some other form.

The file is a line-oriented text file. Blank lines and lines beginning with "#" (comments) will be ignored. Other lines begin with an *entry identifier* like `E` for *Event* or `BS` for *BeginSend*, followed by *attribute-identifier* and *value* pairs. One exception is debug output (recorded from `ev«...` statements), which are represented by lines that begin with a hyphen, and continue with the actual text.

The grammar of the eventlog file is the following:

```
<file> ::= <line>*
<line> ::= <empty-line> | <user-log-message> | <event-log-entry>
<empty-line> ::= CR LF
<user-log-message> ::= - SPACE <text> CR LF
<event-log-entry> ::= <event-log-entry-type> SPACE <parameters> CR LF
<event-log-entry-type> ::= SB | SE | BU | MB | ME | MC | MD | MR | GC | GD |
                           CC | CD | CS | MS | CE | BS | ES | SD | SH | DM | E
<parameters> ::= (<parameter>)*
<parameter> ::= <name> SPACE <value>
<name> ::= <text>
<value> ::= <boolean> | <integer> | <text> | <quoted-text>
```

The eventlog file must also fulfill the following requirements:

- simulation events are in increasing event number and simulation time order

Here is a fragment of an existing eventlog file as an example:

```
E # 14 t 1.018454036455 m 8 ce 9 msg 6
BS id 6 tid 6 c cMessage n send/endTx pe 14
ES t 4.840247053855
MS id 8 d t=TRANSMIT,,#808000;i=device/pc_s
MS id 8 d t=,,#808000;i=device/pc_s
```

[1] With certain granularity of course, and subject to filters that were active during simulation

```
E # 15 t 1.025727827674 m 2 ce 13 msg 25
- another frame arrived while receiving -- collision!
CE id 0 pe 12
BS id 0 tid 0 c cMessage n end-reception pe 15
ES t 1.12489449434
BU id 2 txt "Collision! (3 frames)"
DM id 25 pe 15
```

I.1 Supported Entry Types and Their Attributes

The following entries and attributes are supported in the eventlog file:
SB *(SimulationBegin)*: mandatory first line of an eventlog file

- **v** (*version*, int): OMNeT++ version, e.g. 0x401 (=1025) is release 4.1
- **rid** (*runId*, string): identifies the simulation run
- **b** (*keyframeBlockSize*, int): the distance between keyframes in event numbers

SE *(SimulationEnd)*: optional last line of an eventlog file

- **e** (*isError*, bool): specifies if the simulation terminated due to an error
- **c** (*resultCode*, int): the error code in case of an error, otherwise the normal result code
- **m** (*message*, string): human readable description

BU *(Bubble)*: display a bubble message

- **id** (*moduleId*, int): id of the module which printed the bubble message
- **txt** (*text*, string): displayed message text

MB *(ModuleMethodBegin)*: beginning of a call to another module

- **sm** (*fromModuleId*, int): id of the caller module
- **tm** (*toModuleId*, int): id of the module being called
- **m** (*method*, string): C++ method name

ME *(ModuleMethodEnd)*: end of a call to another module

- no parameters

MC *(ModuleCreated)*: creating a module

- **id** (*moduleId*, int): id of the new module
- **c** (*moduleClassName*, string): C++ class name of the module

- **t** (*nedTypeName*, string): fully qualified NED type name
- **pid** (*parentModuleId*, int): id of the parent module
- **n** (*fullName*, string): full dotted hierarchical module name
- **cm** (*compoundModule*, bool): whether module is a simple or compound module

MD *(ModuleDeleted)*: deleting a module

- **id** (*moduleId*, int): id of the module being deleted

MR *(ModuleReparented)*: reparenting a module

- **id** (*moduleId*, int): id of the module being reparented
- **p** (*newParentModuleId*, int): id of the new parent module

GC *(GateCreated)*: gate created

- **m** (*moduleId*, int): module in which the gate was created
- **g** (*gateId*, int): id of the new gate
- **n** (*name*, string): gate name
- **i** (*index*, int): gate index if vector, -1 otherwise
- **o** (*isOutput*, bool): whether the gate is input or output

GD *(GateDeleted)*: gate deleted

- **m** (*moduleId*, int): module in which the gate was created
- **g** (*gateId*, int): id of the deleted gate

CC *(ConnectionCreated)*: creating a connection

- **sm** (*sourceModuleId*, int): id of the source module identifying the connection
- **sg** (*sourceGateId*, int): id of the gate at the source module identifying the connection
- **dm** (*destModuleId*, int): id of the destination module
- **dg** (*destGateId*, int): id of the gate at the destination module

CD *(ConnectionDeleted)*: deleting a connection

- **sm** (*sourceModuleId*, int): id of the source module identifying the connection
- **sg** (*sourceGateId*, int): id of the gate at the source module identifying the connection

CS *(ConnectionDisplayStringChanged)*: a connection display string change

- **sm** (*sourceModuleId*, int): id of the source module identifying the connection

- **sg** (*sourceGateId*, int): id of the gate at the source module identifying the connection
- **d** (*displayString*, string): the new display string

MS *(ModuleDisplayStringChanged)*: a module display string change

- **id** (*moduleId*, int): id of the module
- **d** (*displayString*, string): the new display string

E *(Event)*: an event that is processing a message

- **#** (*eventNumber*, eventnumber_t): unique event number
- **t** (*simulationTime*, simtime_t): simulation time when the event occurred
- **m** (*moduleId*, int): id of the processing module
- **ce** (*causeEventNumber*, eventnumber_t): event number from which the message being processed was sent, or -1 if the message was sent from initialize
- **msg** (*messageId*, long): lifetime-unique id of the message being processed

KF *(Keyframe)*:

- **p** (*previousKeyframeFileOffset*, int64): file offset of the previous keyframe entry
- **c** (*consequenceLookaheadLimits*, string): consequence lookahead data
- **s** (*simulationStateEntries*, string): simulation state data

abstract *(Message)*: base class for entries referring to a message

- **id** (*messageId*, long): lifetime-unique id of the message
- **tid** (*messageTreeId*, long): id of the message inherited by dup
- **eid** (*messageEncapsulationId*, long): id of the message inherited by encapsulation
- **etid** (*messageEncapsulationTreeId*, long): id of the message inherited by both dup and encapsulation
- **c** (*messageClassName*, string): C++ class name of the message
- **n** (*messageName*, string): message name
- **k** (*messageKind*, short): message kind
- **p** (*messagePriority*, short): message priority
- **l** (*messageLength*, int64): message length in bits
- **er** (*hasBitError*, bool): true indicates that the message has bit errors
- **d** (*detail*, string): detailed information of message content when recording message data is turned on

- **pe** (*previousEventNumber*, eventnumber_t): event number from which the message being cloned was sent, or -1 if the message was sent from initialize

CE *(CancelEvent)*: canceling an event caused by a self message

- no parameters

BS *(BeginSend)*: beginning to send a message

- no parameters

ES *(EndSend)*: prediction of the arrival of a message

- **t** (*arrivalTime*, simtime_t): when the message will arrive to its destination module
- **is** (*isReceptionStart*, bool): true indicates the message arrives with the first bit

SD *(SendDirect)*: sending a message directly to a destination gate

- **sm** (*senderModuleId*, int): id of the source module from which the message is being sent
- **dm** (*destModuleId*, int): id of the destination module to which the message is being sent
- **dg** (*destGateId*, int): id of the gate at the destination module to which the message is being sent
- **pd** (*propagationDelay*, simtime_t): propagation delay as the message is propagated through the connection
- **td** (*transmissionDelay*, simtime_t): transmission duration as the whole message is sent from the source gate

SH *(SendHop)*: sending a message through a connection identified by its source module and gate id

- **sm** (*senderModuleId*, int): id of the source module from which the message is being sent
- **sg** (*senderGateId*, int): id of the gate at the source module from which the message is being sent
- **pd** (*propagationDelay*, simtime_t): propagation delay as the message is propagated through the connection
- **td** (*transmissionDelay*, simtime_t): transmission duration as the whole message is sent from the source gate

CM *(CreateMessage)*: creating a message

- no parameters

CL *(CloneMessage)*: cloning a message either via the copy constructor or dup

- **cid** (*cloneId*, long): lifetime-unique id of the clone

DM *(DeleteMessage)*: deleting a message

- no parameters

References

[BCH+96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 69–82, New York, NY, USA, 1996. ACM.

[BT00] R. L. Bagrodia and M. Takai. Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages. 11(4):395–414, 2000.

[CM79] M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, (5):440–452, 1979.

[EHW02] K. Entacher, B. Hechenleitner, and S. Wegenkittl. A Simple OMNeT++ Queuing Experiment Using Parallel Streams. *PARALLEL NUMERICS'02 - Theory and Applications*, pages 89–105, 2002. Editors: R. Trobec, P. Zinterhof, M. Vajtersic and A. Uhl.

[EPM99] G. Ewing, K. Pawlikowski, and D. McNickle. Akaroa2: Exploiting Network Computing by Distributing Stochastic Simulation. In *Proceedings of the European Simulation Multiconference ESM'99, Warsaw, June 1999*, pages 175–181. International Society for Computer Simulation, 1999.

[For94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. 8(3/4):165–414, 1994.

[Gol91] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[Hel98] P. Hellekalek. Don't Trust Parallel Monte Carlo. *ACM SIGSIM Simulation Digest*, 28(1):82–89, jul 1998. Author's page is a great source of information, see `http://random.mat.sbg.ac.at/`.

[HPvdL95] Jan Heijmans, Alex Paalvast, and Robert van der Leij. Network Simulation Using the JAR Compiler for the OMNeT++ Simulation System. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, 1991.

[JC85] Raj Jain and Imrich Chlamtac. The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms without Storing Observations. *Communications of the ACM*, 28(10):1076–1085, 1985.

[Kof95] Stig Kofoed. Portable Multitasking in C++. *Dr. Dobb's Journal*, November 1995. Download source from `http://www.ddj.com/ftp/1995/1995.11/mtask.zip/`.

[LAM] LAM-MPI home page. `http://www.lam-mpi.org/`.

[Len94] Gábor Lencse. Graphical Network Editor for OMNeT++. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[LSCK02] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An Objected-Oriented Random-Number Package with Many Long Streams and Substreams. *Operations Research*, 50(6):1073–1075, 2002. Source code can be downloaded from `http://www.iro.umontreal.ca/~lecuyer/papers.html`.

[MN98] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudorandom Number Generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998. Source code can be downloaded from `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[MvMvdW95] André Maurits, George van Montfort, and Gerard van de Weerd. OMNeT++ Extensions and Examples. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[OF00] Hong Ong and Paul A. Farrell. Performance Comparison of LAM/MPI, MPICH and MVICH on a Linux Cluster Connected by a Gigabit Ethernet Network. In *Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, October 10-14, 2000*. The USENIX Association, 2000.

[PFS86] Bratley P., B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, 1986.

[PJL02] K. Pawlikowski, H. Jeong, and J. Lee. On Credibility of Simulation Studies of Telecommunication Networks. *IEEE Communications Magazine*, pages 132–139, jan 2002.

[Pon91] György Pongor. OMNET: An Object-Oriented Network Simulator. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1991.

[Pon92] György Pongor. Statistical Synchronization: A Different Approach of Parallel Discrete Event Simulation. Technical report, University of Technology, Data Communications Laboratory, Lappeenranta, Finland, 1992.

[Pon93] György Pongor. On the Efficiency of the Statistical Synchronization Method. In *Proceedings of the European Simulation Symposium (ESS'93), Delft, The Netherlands, Oct. 25-28, 1993*. International Society for Computer Simulation, 1993.

[Qua] Quadrics home page. `http://www.quadrics.com/`.

[ŞVE03] Y. Ahmet Şekercioğlu, András Varga, and Gregory K. Egan. Parallel Simulation Made Easy with OMNeT++. In *Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands*. International Society for Computer Simulation, 2003.

[Var92] András Varga. OMNeT++ - Portable Simulation Environment in C++. In *Proceedings of the Annual Students' Scientific Conference (TDK), 1992*. Technical University of Budapest, 1992. In Hungarian.

[Var94] András Varga. Portable User Interface for the OMNeT++ Simulation System. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[Var98a] András Varga. K-split – On-Line Density Estimation for Simulation Result Collection. In *Proceedings of the European Simulation Symposium (ESS'98), Nottingham, UK, October 26-28*. International Society for Computer Simulation, 1998.

[Var98b] András Varga. Parameterized Topologies for Simulation Programs. In *Proceedings of the Western Multiconference on Simulation (WMC'98) Communication Networks and Distributed Systems (CNDS'98), San Diego, CA, January 11-14*. International Society for Computer Simulation, 1998.

[Var99] András Varga. Using the OMNeT++ Discrete Event Simulation System in Education. *IEEE Transactions on Education*, 42(4):372, November 1999. (on CD-ROM issue; journal contains abstract).

[Vas96] Zoltán Vass. PVM Extension of OMNeT++ to Support Statistical Synchronization. Master's thesis, Technical University of Budapest, 1996. In Hungarian.

[VF97] András Varga and Babak Fakhamzadeh. The K-Split Algorithm for the PDF Approximation of Multi-Dimensional Empirical Distributions without Storing Observations. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997*, pages 94–98. International Society for Computer Simulation, 1997.

[VP97] András Varga and György Pongor. Flexible Topology Description Language for Simulation Programs. In *Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997*, pages 225–229, 1997.

[VŞE03] András Varga, Y. Ahmet Şekercioğlu, and Gregory K. Egan. A practical efficiency criterion for the null message algorithm. In *Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands*. International Society for Computer Simulation, 2003.

[Wel95] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, 1995.

Index

abstract, 137, 143
acceptDefault(), 304
accuracy detection, 176
activity(), 51–53, 56–58, 61–65, 84, 97, 179, 251, 253
addBinBound(), 173
addGate(), 74
addObject(), 127
addPar(), 127
addParametersAndGatesTo(module), 303
Akaroa, 248
allowunconnected, 29, 35, 36, 319
any, 184
arrival time, 48, 84
asDoubleVector(), 70
asIntVector(), 70
asVector(), 70

back(), 160
backgroundDisplayString(), 211
beginSend(), 305
bernoulli(p, *rng=0*), 159
beta(alpha1, alpha2, *rng=0*), 158
binary heap, 50
binary tree, 35
binomial(n, p, *rng=0*), 159
bit error, 84
bool, 22, 131, 184, 315
boolValue(), 67, 128, 185
bubble(), 212
buildInside(), 98, 99, 303
buildInside(module), 303

cAccuracyDetection, 176
cADByStddev, 176
calculateUnweightedSingleShortestPathsTo(), 165
callFinish(), 99
callInitialize(), 56, 98
cancelAndDelete(), 77
cancelAndDelete(msg), 54
cancelEvent(), 57, 62, 77, 78, 123
cArray, 127, 151, 154, 161, 162, 195–197
cauchy(a, b, *rng=0*), 158
cChannel, 50, 76, 86, 87, 102, 312
cChannelType, 100, 303
cComponent, 50, 51, 53, 67, 86, 102, 212
cConfigOption, 293
cConfiguration, 293, 302
cConfigurationEx, 289, 290
cConfigurationReader, 290
cDatarateChannel, 50, 76, 86, 88, 89, 100
cDelayChannel, 50, 76, 86, 88, 100
cDensityEstBase, 170
cDisplayString, 205, 211
cDoubleHistogram, 151, 159, 169, 170, 256
cDynamicExpression, 69
ceil(), 49
cEnvir, 151, 293, 301, 303–306
cExpression, 69
cFileOutputScalarManager, 289, 292
cFileSnapshotManager, 289, 292
cFSM, 90, 91
cGate, 21, 71, 72, 74–76, 96, 100, 101
chain, 34
Channel, 86
channel, 312, 316
channelinterface, 312
char, 131
check-signals, 104
check_and_cast<>(), 96, 304
chi_square(k, *rng=0*), 158
cIdealChannel, 50, 76, 86, 88, 100
cIListener, 105–107
cIndexedFileOutputVectorManager, 259, 289, 292
cITimestampedValue, 117
cKSplit, 151, 159, 169, 170, 172, 175, 256
class, 130, 136, 138, 149
cLinkDelayLookahead, 285
cListener, 107
cLongHistogram, 151, 159, 170, 256
cMatchableString, 169
cMatchExpression, 166, 168, 169

cMatchExpression::Matchable, 168
Cmdenv, 155, 243
cmdenv-express-mode, 242, 244
cmdenv-interactive, 242
cmdenv-performance-display, 244
cmdenv-runs-to-execute, 237, 244
cmdenv-status-frequency, 242
cMessage, 48, 76, 78, 88, 119–124, 126, 127, 129, 130, 135, 136, 149, 151, 159, 194, 196, 207, 208, 375
cMessageHeap, 50
cModelChangeNotification, 108
cModule, 50, 55, 71, 73, 74, 90, 95, 96, 99, 102, 151, 158, 304, 313
cModuleType, 97, 98, 303
cMsgPar, 127
cMySQLOutputScalarManager, 261
cNamedObject, 121, 138, 153
cNEDValue, 185–187
cNMPLookahead, 285
cNullEnvir, 301
cNullMessageProtocol, 285
cNumericResultFilter, 117
cNumericResultRecorder, 117
cObject, 105, 108, 119, 122, 127, 136, 138, 139, 149, 152, 153, 159–161, 180, 189, 194
cObjectFactory, 104
cObjectResultFilter, 117
collect(), 170
command line options, 246
command line user interface, 243
configuration-class, 290
connect(), 100
connection, 5
  creating, 100
  removing, 101
connections, 16, 32, 319
connectTo(), 100
const, 315
constraint, 231, 233
convertTo(), 187
convertUnit(), 187
copy(), 190, 192, 193
coroutine, 53, 61, 63
  stack size, 65
cout, 155
cOutputScalarManager, 289, 292
cOutputVectorManager, 264, 289, 292
cOutVector, 116, 151, 177, 250, 256, 259, 262, 289, 292
cOwnedObject, 137–139, 152, 189–192, 194–196, 253
cPacket, 76, 81, 87, 88, 111, 119–122, 124–126, 129, 130, 135, 136, 149, 152, 375
cPar, 67, 69, 151, 159, 172
cParsimCommunications, 285
cParsimSynchronizer, 285
cPatternMatcher, 166–168
cplusplus, 137, 139, 146, 147
cPostDisplayStringChangeNotification, 108
cPostGateAddNotification, 108
cPostGateConnectNotification, 108
cPostGateDeleteNotification, 108
cPostGateDisconnectNotification, 108
cPostGateVectorResizeNotification, 108
cPostModuleAddNotification, 108
cPostModuleDeleteNotification, 108
cPostModuleReparentNotification, 108
cPostParameterChangeNotification, 108
cPostPathCreateNotification, 108
cPostPathCutNotification, 108
cPreDisplayStringChangeNotification, 108
cPreGateAddNotification, 108
cPreGateConnectNotification, 108
cPreGateDeleteNotification, 108
cPreGateDisconnectNotification, 108
cPreGateVectorResizeNotification, 108
cPreModuleAddNotification, 108
cPreModuleDeleteNotification, 108
cPreModuleReparentNotification, 108
cPreParameterChangeNotification, 108
cPrePathCreateNotification, 108
cPrePathCutNotification, 108
cProperties, 69
cPSquare, 151, 159, 169–172, 256
CPU time, 48
cpu-time-limit, 224, 242, 247
cQueue, 85, 151, 154, 159, 160, 189, 194–197, 204
cQueue::Iterator, 161
cRealTimeScheduler, 290, 306
create(), 98–100
createModuleObject(), 303
createOne(), 190, 191
createScheduleInit(), 98
cResultFilter, 117
cResultRecorder, 117
cRNG, 157, 158, 289, 290
cRunnableEnvir, 294
cRuntimeError, 186, 301, 305
cScheduler, 289, 290

cSequentialScheduler, 305, 306
cSimpleModule, 18, 50, 52, 53, 56, 84, 99, 178, 311
cSimulation, 95, 298, 301, 303–306
cSimulation::setActiveSimulation(), 306
cSnapshotManager, 289, 292
cStaticFlag, 299
cStatictic, 262
cStatistic, 169, 176, 178
cStdDev, 151, 169, 170, 172, 256, 262
cStringPool, 187
cStringTokenizer, 70
cSubModIterator, 95
cTDExpandingWindows, 176
cTerminationException, 301, 305
cTimestampedValue, 116
cTopology, 151, 162–165
cTopology::Link, 163, 164
cTopology::LinkIn, 164
cTopology::LinkOut, 163–165
cTopology::Node, 163–165
cTransientDetection, 176
customization, 297
cVarHistogram, 151, 159, 169, 170, 173
cWeightedStdDev, 169, 170, 256
cWeightedStddev, 151

dbl(), 49
dblrand(), 157, 158
debug-on-errors, 242
debug-statistics-recording, 114
decapsulate(), 125, 126
Define_Channel(), 86
Define_Function(), 183
Define_Module(), 52, 86, 97, 299
Define_NED_Function(), 159, 183, 185, 189
Define_NED_Math_Function(), 183, 188
delayed sending, 80
deleteGate(), 74
deleteModule(), 99, 109
deleteNetwork(), 301
detect(), 176
Dijkstra algorithm, 165
disable(), 165
disconnect(), 101
discrete event simulation, 47
display strings, 199
displayString, 207
distribution
  as histogram, 159
  custom, 159, 172
  estimation, 170
  even, 174
  multi-dimensional, 173
  online estimation, 173
  predefined, 158
  proportional, 174
  random variables, 158
dlopen(), 241
doneLoadingNedFiles(), 303
doOneEvent, 305
doOneEvent(), 301
double, 22, 131, 184, 187, 191, 315
doubleValue(), 67, 128, 185
doubleValueInUnit(), 186, 187
drop(), 126, 196
dup(), 105, 121, 126, 130, 142, 154, 196

embedding, 297
emit(), 102, 103, 105, 107, 116
emit(simsignal_t, cObject *), 116
empty(), 160
enable(), 165
encapsulate(), 125
end(), 62, 161
end-of-simulation, 56
endRun(), 301
endSend(), 305
endSimulation(), 89
entry code, 90
enum, 132
EnvirBase, 294
envirbase.h, 294
erlang_k(k, mean, *rng=0*), 158
error(), 90
EV, 155
ev, 91, 151, 155, 306
ev.addResultRecorders(), 115
ev.printf(), 155
event, 56, 64
  causality, 279
event loop, 54, 63
event timestamp, 48
events, 47, 48
  initial, 58
exit code, 90
experiment-label, 235
exponential(), 49, 151
exponential(mean, *rng=0*), 158
extends, 42, 135, 136, 138, 139, 224, 225, 231
extraStackforEnvir, 183

fabs(), 49

false, 331
FEL, 48
FES, 48–50, 54, 64, 77, 84, 87, 98, 119, 194, 244, 245, 287
finalize(), 56
finalizeParameters(), 98
findGate(), 72
findIncomingTransmissionChannel(), 76
findSubmodule(), 95
findTransmissionChannel(), 76
fingerprint, 242
finish(), 51, 53–56, 58, 65, 71, 86, 99, 107, 178, 183, 244, 251, 301
finite state machine, 61, 90
float, 131
floor(), 49
fmod(), 49
fname-append-host, 251
for, 16, 323, 331, 332
for(), 92
forceTransmissionFinishTime(), 84
forEachChild(), 189, 190, 193, 194
front(), 160
FSM, 61, 90, 92
  nested, 90
FSM_DEBUG, 91
FSM_Goto(), 91
FSM_Print(), 92
FSM_Switch(), 90–92
future events, 48

gamma_d(alpha, beta, *rng=0*), 158
gate, 5, 71
gate(), 71–73
gateBaseId(), 73
gateHalf(), 72
GateIterator, 73
gates, 17, 317
gateSize(), 72
gateType(name), 73
geometric(p, *rng=0*), 159
getArrivalTime(), 123
getAsBool(), 302
getAsInt(), 302
getBaseClassDescriptor(), 149
getBaseName(), 74
getBasepoint(), 171
getByteLength(), 111, 124
getCDF(), 172
getCellInfo(), 171
getCellPDF(), 171
getCellValue(), 171
getChannel(), 76
getClassName(), 152, 153
getConfig(), 293
getConfigValue(), 302
getCount(), 170
getCreationTime(), 121
getDefaultOwner(), 196
getDeliverOnReceptionStart(), 83, 84, 87
getDisplayString(), 123, 207, 211
getDistanceToTarget(), 165
getDuration(), 82, 84
getEncapsulatedPacket(), 125, 126
getFieldAsString(), 149
getFieldCount(), 149
getFieldName(), 149
getFieldTypeString(), 149
getFullName(), 74, 153
getFullPath(), 74, 153
getGateNames(), 73
getId(), 72, 74, 94
getIncomingTransmissionChannel(), 76
getIndex(), 74, 94
getKind(), 208
getLocalGate(), 164
getLocalGateId(), 164
getLocalListenedSignals(), 106
getLocalSignalListeners(), 106
getMax(), 170
getMaxTime, 49
getMean(), 170
getMin(), 170
getModuleByPath(), 95, 304
getModuleByRelativePath(), 95
getName(), 69, 74, 95, 121, 152, 153
getNameSuffix(), 74
getNextGate(), 75, 87, 96
getNodeFor(), 164
getNumCells(), 171
getNumInLinks(), 164
getNumNodes(), 164
getNumOutLinks(), 164
getNumPaths(), 165
getNumRNGs(), 301
getObject(), 127
getOverflowCell(), 171
getOwner(), 194, 195
getOwnerModule(), 74, 96
getParentModule(), 86, 95, 96
getParList(), 127
getPathEndGate(), 75
getPathStartGate(), 75
getPDF(), 172

getPooled(), 187
getPreviousGate(), 75, 96
getProperties(), 69
getRemoteGate(), 164
getRemoteGateId(), 164
getRemoteNode(), 164
getRNG(), 158
getRNG(k), 301
getScaleExp(), 49
getSendingTime(), 123
getSignalName(), 103
getSignalTime(), 117
getSignalValue(), 117
getSourceGate(), 76
getSqrSum(), 170
getStackUsage(), 183, 251
getStddev(), 170
getSubmodule(), 95, 96
getSum(), 170
getTargetNode(), 165
getTransmissionChannel(), 76, 82
getTransmissionFinishTime(), 82, 83, 86–88
getType(), 69, 74, 187
getTypeName(), 69, 187
getUnderflowCell(), 171
getUnit(), 69, 186, 187
getVariance(), 170
getVectorSize(), 74, 94
getWarmupPeriod(), 258
global variables, 66
GNU R, 264
Gnuplot, 265
Grace, 266

handleMessage(), 51–53, 56–59, 61, 63, 77, 84, 85, 87, 90, 177, 253
handleParameterChange, 56
handleParameterChange(), 69–71, 88, 89
hasBitError() method, 83
hasGate(), 72
hasListeners(), 103
hasMoreTokens(), 70
hasObject(), 127
hasPar(), 128
histogram
  equal-sized, 170
  equiprobable-cells, 170
  range estimation, 171

if, 31, 319, 323
import, 44
index, 26, 332
info(), 194
ini file
  file inclusion, 223
InifileReader, 290, 291
initial events, 48
initialization, 55
  multi-stage, 55
initialize(), 51, 53–59, 61, 63–65, 68, 71, 83, 86, 97, 98, 103, 177, 179, 301
initialize(int stage), 56
inout, 317, 320
input, 317, 318, 320
insert(), 159, 160
insertAfter(), 160
insertBefore(), 160
int, 22, 131, 132, 143, 184, 186, 315
int16_t, 131
int32_t, 131
int64_t, 131
int8_t, 131
intrand(), 158
intrand(n), 157
intuniform(), 159
intuniform(a, b, *rng=0*), 159
isBusy(), 82, 83, 87
isConnected(), 75
isConnectedInside(), 75
isConnectedOutside(), 75
isEnabled(), 165
isGateVector(name), 73
isInstance(), 104
isNumeric(), 69, 187
isPacket(), 88, 121
isReceptionStart(), 83
isScheduled(), 77, 123
isSelfMessage(), 77, 123
isSet(), 187
isSubscribed(), 106
isTransformed(), 171
isTransmissionChannel(), 76, 86–88
isVector(), 74
isVolatile(), 69
iter(), 95

length(), 160
like, 37, 42, 45, 319, 322, 326, 329
link, 5
load-libs, 223, 241
loadFromFile(), 172
LoadLibrary(), 241
loadNedFile(), 303
loadNedSourceFolder(), 303

loadNedText(), 303
lognormal(m, s, rng=0), 158
long, 131
LongHistogram, 169
longValue(), 67, 128, 185

main(), 297, 299
make, 216, 217, 248
Makefile, 216
    dependencies, 217
matches(), 166
Matlab, 265
MatPlotLib, 265
MAXTIME, 50
mayHaveListeners(), 103
measurement-label, 235
message, 48, 130, 136, 149
    attaching non-object types, 127
    attaching objects, 127
    cancelling, 77
    duplication, 121
    exchanging, 5
    IDs, 122
    priority, 49
message definitions, 213
method calls
    between modules, 96
model
    time, 48
module, 311
    accessing parameters, 67
    compound, 4
      patterns, 36
    constructor, 53, 57
    destructor, 55
    dynamic creation, 97
    dynamic deletion, 99
    hierarchy, 4
    id, 94
    libraries, 5, 8
    parameters, 5
    simple, 2, 4, 6, 47, 51, 52, 64, 279
    stack size, 53, 182
    submodule
      lookup, 95
    types, 4
    vector, 94
      iteration, 95
Module_Class_Members(), 53
moduleByRelativePath(), 95
moduleinterface, 312
msg
    files, 213
Multiple Replications in Parallel, 248
MultiShortestPathsTo(), 165
multitasking
    cooperative, 63

ned
    expressions, 331
      operators, 331
    files, 6, 7, 213
    functions, 332
    language, 2, 335
ned-path, 43, 240
NEDFunction, 185
nedtool, 303
negbinomial(n, p, *rng=0*), 159
network, 14, 224, 311
nextToken(), 70
noncobject, 138
noncopyable, 105
normal(), 49, 151
normal(mean, stddev, *rng=0*), 158
num-rngs, 233, 236
numInitStages(), 56
NumPy, 265

object
    copy, 154
    duplication, 154
objectValue(), 128
Octave, 265
omnetpp.ini, 7, 14, 17, 22, 23, 25, 27, 92, 155–157, 177, 190, 209, 221, 224, 226, 236, 237, 239, 244, 250, 251, 289, 290, 324, 327, 392
omnetpp.sna, 181
operator=(), 126, 130, 142, 154, 197
opp_error(), 154
opp_lcg32_seedtool, 238
opp_makedep, 217
opp_makedep --help, 217
opp_makemake, 215–219, 239, 240
opp_msgc, 130, 218
opp_neddoc, 276
opp_run, 184, 239, 242
opp_runall, 229, 247, 248
optimal routes, 162
optimal routing, 164
output, 317, 320
    file, 216
    gate, 78
    scalar file, 7

scalars, 178
vector file, 7
vector object, 178
output-scalar-file, 223, 258
output-scalar-precision, 260
output-vector-file, 223, 258
output-vector-precision, 260
outputscalarmanager-class, 292
outputvectormanager-class, 261, 292
ownership, 79, 152, 196

package, 310
package.ned, 43, 44, 310
packet, 130, 136, 149
encapsulation, 125
par(), 67, 128
parallel simulation, 279
conservative, 279
optimistic, 279
parallel-simulation, 285
parameters, *see* module parameters, 17, 315, 316, 324, 325, 327, 328
pareto_shifted(a, b, c, *rng=0*), 158
parse(), 49, 304
PARSEC, 56
parseQuantity(), 187
parsim-communications-class, 285
parsim-debug, 285
parsim-nullmessageprotocol-laziness, 285
parsim-nullmessageprotocol-lookahead-class, 285
parsim-synchronization-class, 285
parsimPack(), 190
parsimUnpack(), 190
path(), 165
PDES, 279
pointerValue(), 128
poisson(lambda, *rng=0*), 159
pop(), 159, 160
POST_MODEL_CHANGE, 108, 109
PRE_MODEL_CHANGE, 108, 109
printf(), 90, 155
processMessage(), 86–89
property, 315

quantity, 184, 187
queue
iteration, 161
order, 160
queue.setName(procqueue), 204

random
numbers, 172
numbers from distributions, 158
random number generator, 156
random(), 172
raw(), 49
readParameter(), 301, 303
real time, 48
receive
timeout, 85
receive(), 51, 57, 62–64, 84, 85, 87
receiveSignal(), 102, 107
record(), 177, 178
record-eventlog, 242
recordScalar(), 256–258, 301, 304
recordStatistic(), 301
recordWithTimestamp(), 116, 177
Register_Abstract_Class(), 104
Register_Class(), 104, 190, 299
Register_Class(MyRNGClass), 289
Register_OmnetApp(), 294
Register_ResultFilter(NAME, CLASSNAME), 117
Register_ResultRecorder(NAME, CLASSNAME), 117
registerSignal(), 102, 103, 105, 115
remove(), 160, 162, 196
removeObject(), 127
repeat, 233, 234
replication-label, 235
result accuracy, 176
result-dir, 258
result-recording-modes, 256
result_t, 87, 88
results/Fifo1-0.vec, 244
rng-class, 236, 290
ROOT, 265
routing support, 162

saveToFile(), 173
scalar-recording, 257
scavetool, 263–266
scheduleAt(), 57, 62, 77, 80, 85, 123, 195
scheduler-class, 290
scheduleStart(), 98
SciPy, 265
SectionBasedConfiguration, 290, 291
seed-set, 233, 234
seedtool, 157
segmentation fault, 252
selectNextModule(), 301, 305
self-message, 57, 77
cancelling, 77
send(), 57, 62, 79, 80, 84, 87, 195
send...(), 123

sendDelayed(), 80
sendDirect(), 29, 81, 82, 84, 318
sendHop(), 305
set(), 185, 187
set...ArraySize(), 133, 134
setActiveSimulation(NULL), 301
setBitError(), 88
setBitErrorRate(), 100
setBoolValue(), 69, 127, 304
setControlInfo(), 122, 136
setDatarate(), 100
setDelay(), 100
setDeliverOnReceptionStart(), 21
setDeliverOnReceptionStart(), 82, 83
setDisplayString(), 202, 203, 207
setDoubleValue(), 69, 127
setDuration(), 87, 88
setFieldAsString(), 149
setGateSize(), 74
setLongValue(), 69, 127, 304
setName(), 121, 153, 154
setObjectValue(), 127
setPacketErrorRate(), 100
setPattern(), 166–168
setPointerValue(), 127
setPreservingUnit(), 186
setPreservingUnit(double), 187
setScheduler(), 306
setStringValue(), 69, 127
setTagArg(), 205
setTimeStamp(), 121
setUnit(), 187
setupGateVectors(module), 303
setupNetwork(), 301
setXMLValue(), 69, 127
short, 131, 143
shortest path, 162
sim-time-limit, 224, 242, 247
simple, 14, 17, 311
simTime(), 77, 177
SIMTIME_DBL(t), 50
SIMTIME_RAW(t), 50
SIMTIME_STR(), 50
simtime_t, 49, 131, 155, 156
SIMTIME_TTOA(buf,t), 50
simtimeToStr(), 155, 156
simtimeToStrShort(), 156
simulation, 298, 306
  building, 6
  concepts, 47
  configuration file, 7
  debugging, 246
  kernel, 7, 214, 297
  running, 6
  user interface, 7
simulation time, 48
simulation time limits, 89
SingleShortestPaths(), 165
size(), 74, 162
sizeof, 26
sizeof(), 332
skiplist, 50
snapshot file, 179, 181
snapshot(), 181, 190
snapshotmanager-class, 292
Spreadsheets, 266
sprintf(), 153
sputn(), 301
stack, 63
  for Tkenv, 253
  overflow, 65, 183
  size, 53, 182, 251
  too small, 251
  usage, 65
  violation, 183
starter messages, 54, 57, 64, 97
startRun(), 301
state transition, 91
std::exception, 301, 305
std::string detailedInfo(), 190
std::string info(), 190
stdstringValue(), 67, 185
steady states, 90
sTopoLinkIn, 163
str(), 49, 69
STR_SIMTIME(s), 50
string, 22, 37, 132, 137, 184, 315
stringValue(), 67, 128, 185
strToSimtime(), 156
strToSimtime0(), 156
struct, 130
student_t(i, *rng=0*), 158
submodules, 16, 318
subscribe(), 105
subscribedTo(), 107
suspend execution, 62
switch(), 92

take(), 126, 192
this, 332
Tkenv, 151, 246
tkenv-default-config, 246
tkenv-default-run, 246
tkenv-extra-stack, 246

tkenv-image-path, 246
tkenv-plugin-path, 246
topology
    description, 6
    hypercube, 36
    patterns, 36
    random, 35
    shortest path, 164
    tree, 36
total-stack, 251
transferTo(), 63
transform(), 171, 173
transient detection, 176
transient states, 90
triang(a, b, c, *rng=0*), 158
true, 331
truncnormal(), 159
truncnormal(mean, stddev, *rng=0*), 158
typename, 38
types, 323

uint16_t, 131
uint32_t, 131
uint64_t, 131
uint8_t, 131
ulimit, 251, 252
uniform(), 49
uniform(a, b, *rng=0*), 158
unsigned char, 131
unsigned int, 131
unsigned long, 131
unsigned short, 131
unsubscribe(), 105, 107
unsubscribedFrom(), 107
user interface, 7
user-interface, 294

valgrind, 242
vector-record-eventnumbers, 259
vector-recording, 257, 259
vector-recording-intervals, 257, 259
virtual, 131
virtual time, 48
volatile, 22, 26, 315

wait(), 57, 62–64, 85
waitAndEnqueue(), 85
warmup-period, 257
WATCH(), 151, 179, 180
WATCH_LIST(), 180
WATCH_MAP(), 180
WATCH_OBJ(), 180
WATCH_PTR(), 180
WATCH_PTRLIST(), 180
WATCH_PTRMAP(), 180, 181
WATCH_PTRSET(), 180
WATCH_PTRVECTOR(), 180
WATCH_RW(), 180
WATCH_SET(), 180
WATCH_VECTOR(), 180
weibull(a, b, *rng=0*), 158

X *dup() const, 190
xml, 22, 27, 28, 184, 315, 333
xml(), 28, 333
xmldoc(), 27, 28, 333
xmlValue(), 67, 128, 185

zero stack size, 57