OMNeT++

- User Manual
- API Reference
- IDE User Guide
- Install Guide

omnetpp.org/documentation

F. Xabier Albizuri - 2014

Introduction

OMNeT++ is an object-oriented modular discrete event system simulation framework.

It has a generic architecture, so it can be used in various problem domains: modeling communication networks, protocol modeling, queueing networks, multiprocessors and other distributed hardware 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.

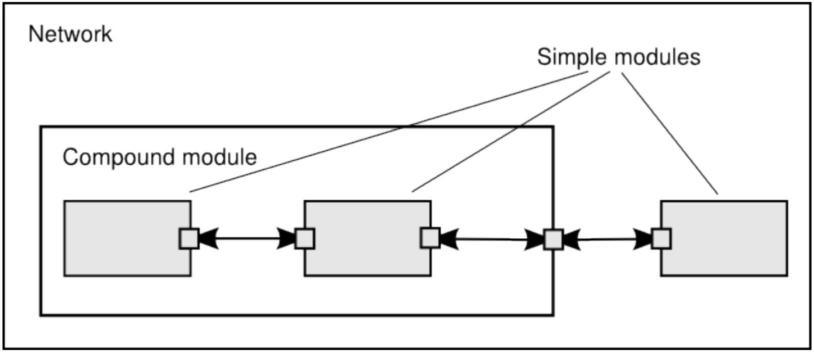
OMNeT++ itself is not a simulator of anything concrete, but rather provides infrastructure and tools for writing simulations. One of the fundamental ingredients of this infrastructure is a component architecture for simulation models.

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.

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.

Modeling Concepts



Modeling Concepts

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*. 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. Because of the hierarchical structure of the model, messages typically travel through a chain of connections, starting and arriving in simple modules.

Modeling Concepts: Hierarchical Modules

An OMNeT++ model consists of hierarchically nested modules that communicate by passing messages to each other. 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 simulation class library.

Modeling Concepts: 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.

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.

Module types can be stored in files separately, the user can group existing module types and create component libraries.

Modeling Concepts: 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.

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.

Modeling Concepts: Messages, Gates, Links

Each connection (also called link) is created within a single place 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.

Because of the hierarchical structure of the model, messages typically travel through a series of connections, starting and arriving in simple modules.

Simple modules can send messages either directly to their destination, or through gates and connections.

Modeling Concepts: 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, 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.

Modeling Concepts: Parameters

Modules can have *parameters*, and they can be assigned in either the NED files or the configuration file omnetpp.ini.

Parameters can be used to customize simple module behavior.

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.

Using OMNeT++: Building and Running Simulations

An OMNeT++ model consists of the following parts:

- NED language topology descriptions (.ned files) that describe the module structure with parameters, gates, etc. NED files can be written using any text editor. The OMNeT++ IDE provides excellent support for 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 (C++ compiled into libraries) provides:

- Simulation kernel. This contains the code that manages the simulation and the simulation class library.
- User interfaces used in simulation execution, to facilitate debugging, demonstration, or batch execution of simulations.

Using OMNeT++: Building and Running Simulations

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.
- NED files are loaded dynamically in their original text forms when the simulation program starts.

The simulation may be compiled as a standalone program executable (for other machines without OMNeT++), or it can be created as a shared library (OMNeT++ shared libraries must be present). 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.

Using OMNeT++: Building and Running Simulations

The output of the simulation is written into result files: output vector files, output scalar files, and possibly the user's own output files. Output files are line-oriented text files. The OMNeT++ IDE provides rich environment for analyzing these files.

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.

The same simulation model can be executed with various user interfaces. 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.

The NED Language: Overview

The user describes the structure of a simulation model in the *NEtwork Description* language.

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.

The NED Language: Overview

Hierarchical. A too complex module can be broken down into smaller modules, and used as a compound module.

Component-Based. Simple modules and compound modules are inherently reusable, which allows component libraries 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.

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.

Packages (Java-like), *Inner types* (channel types and module types defined within a compound module), *Metadata annotations*.

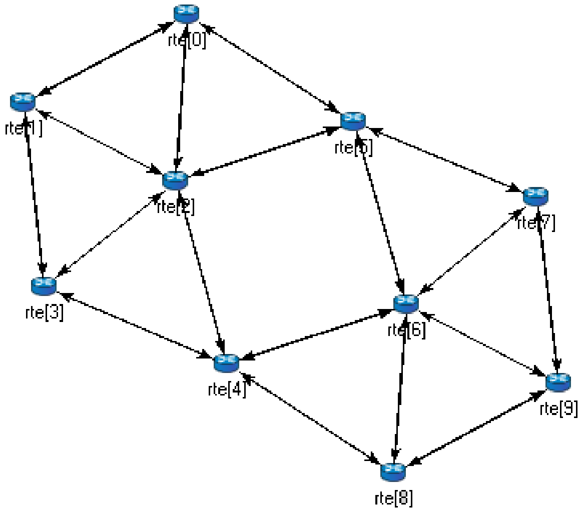
The NED Language: Quickstart

We introduce the NED language via an 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.

The NED Language: Quickstart



The NED Language: Quickstart

```
//
// The NED description of my system
//
network MyNetwork
{
 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 NED Language: Quickstart

The above code defines a network type named MyNetwork. The network contains several nodes, named `node1`, `node2`, etc. from the NED module type Node.

The `port++` notation adds a new gate to the `port[]` gate vector. The double arrow means bidirectional connection. Nodes are connected with a channel that has a data rate of 100Mbps.

The above code would be placed into a NED file, say Net.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 = MyNetwork
```

The NED Language: Quickstart

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. Concepts used: inner types, channels, the DatarateChannel built-in type, inheritance.

```
network MyNetwork
{
  types:
    channel C extends ned.DatarateChannel {
      datarate = 100Mbps;
    }
  submodules:
    ...
  connections:
    node1.port++ <--> C <--> node2.port++;
    node2.port++ <--> C <--> node4.port++;
    ...
}
```

The NED Language: Quickstart

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).

The functionality of the module Node is quite complex, so it is better to implement it with several smaller simple module types which we are going to assemble into a compound module:

- A simple module for traffic generation (App),
- One for routing (Routing),
- And one for queueing up packets to be sent out (Queue).

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

The NED Language: Quickstart

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

simple Routing
{
  ...
}

simple Queue
...
```

The NED Language: Quickstart

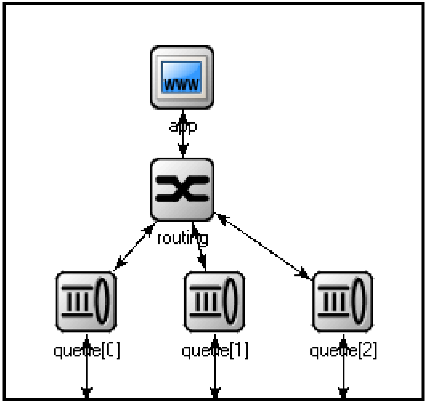
The simple module App has a parameter called `destAddress` (others have been omitted for now), and two gates named `out` and `in` for sending and receiving application packets.

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.

We can assemble App, Routing and Queue into the compound module Node.

The NED Language: Quickstart



The NED Language: Quickstart

```
module Node
{
  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];
```

The NED Language: Quickstart

```
    }
}
```

The compound module Node contains an `address` parameter, plus a gate, named `port`. The type of `port[]` is `inout`, which allows bidirectional connections. It is legal to refer to `sizeof(port)`, the gate vector size will be determined implicitly by the number of neighbours when we create a network from nodes of this type.

The Node module type has an app submodule and a `routing` submodule, plus a `queue[]` submodule vector that contains one Queue module for each `port`.

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).

The NED Language: Quickstart

Putting it together:

- 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 MyNetwork network. Then the network is instantiated for simulation.

The simulation model is built in a *top-down* preorder fashion: 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.

The NED Language: Simple Modules

Simple modules are the model active components. An example (parameters and gates sections are optional):

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

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).

The C++ classes need to be subclassed from cSimpleModule class.

The NED Language: Simple Modules

One can explicitly specify the C++ class with the `@class` property. Classes with C++ namespace qualifiers are also accepted.

```
simple Queue
{
  parameters:
    int capacity;
    @class(mylib::Queue);
    @display("i=block/queue");
  ...
}
```

If you have several modules in a file that are all in a common namespace, then a better alternative to `@class` is the `@namespace` property, specified at the file level (`@namespace(mylib);`). The namespace will be prepended to the normal class name. Moreover, when the `@namespace` property is placed in a file called package.ned, the namespace will apply to all files in the same directory and all directories below.

The NED Language: Simple Modules

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, or to replace the C++ class with a different one. By default, the derived NED module type will inherit the C++ class from its base; you need to write out `@class` if you want it to use the new class.

The following examples show how to specialize a module and how to override the inherited C++ class.

```
simple BoundedQueue extends Queue
{
   capacity = 10;
}
simple PriorityQueue extends Queue
{
   @class(PriorityQueue);
}
```

The NED Language: Compound Modules

A compound module groups other modules into a larger unit. No active behavior is associated with it. A compound module declaration may contain several sections, all of them optional:

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

The NED Language: Compound Modules

Modules contained in a compound module are called submodules. 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 (to not pollute the namespace).

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

The NED Language: Compound Modules

We show a *stub* for wireless hosts. Module types and gate names are fictional (not a real framework). We add user agents via subclassing. The module is further extended with an Ethernet port.

```
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;
}
```

The NED Language: Compound Modules

```
module WirelessHost extends WirelessHostBase
{
  submodules:
    webAgent: WebAgent;
  connections:
    webAgent.tcpOut --> tcp.appIn++;
    webAgent.tcpIn <-- tcp.appOut++;
}
module DesktopHost extends WirelessHost
{
  gates:
    inout ethg;
  submodules:
    eth: EthernetNic;
  connections:
    ip.nicOut++ --> eth.ipIn;
    ip.nicIn++ <-- eth.ipOut;
    eth.phy <--> ethg;
}
```

The NED Language: 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 default class name is the NED type name (unless there is a `@class` property).

There are predefined channel types that you can subclass from. The predefined types are (you can get rid of the package name if you import the types with the `import ned.*` directive):

- ned.IdealChannel
- ned.DelayChannel
- ned.DatarateChannel

The C++ class is inherited when the channel is subclassed.

The NED Language: Channels

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, this type has its uses when a channel object is required.

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.

The NED Language: Channels

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 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.

The NED Language: Channels

A new channel type by specializing DatarateChannel:

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

You may add parameters and properties to channels via subclassing, and may modify existing ones:

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

The NED Language: Channels

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 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);
}
```

The NED Language: 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 be specified (`@unit` property), to increase safety.

Parameters may get their values 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. A default value can also be given (`default()`).

The next example shows a simple module with five parameters.

The NED Language: Parameters

```
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;
}
```

The NED Language: Parameters

We specialize the above App module type via inheritance:

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

The module definition sets the `protocol` parameter to a fixed value, this parameter is now locked, its value cannot be modified via further subclassing or other ways. The default values of the two other parameters are changed.

The NED Language: Parameters

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; this setting cannot be changed.

The NED Language: Parameters

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:

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

The NED Language: Parameters

Parameter assignment can also be placed into the parameters block of the parent module, which provides additional flexibility:

```
network MyNetwork
{
  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.

The NED Language: Parameters

If you had a number of individual hosts instead of a submodule vector:

```
network MyNetwork
{
  parameters:
    host*.ping.timeToLive = default(3);
    host{0..49}.ping.destAddress = default(50);
    host{50..}.ping.destAddress = default(0);
  submodules:
    host0: Host;
    host1: 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.

The NED Language: Parameters

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*.

The double asterisk is one more wildcard that can be used in pattern assignments: it matches any sequence of characters including dots, so it can match multiple path elements.

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

The NED Language: Parameters

A parameter can be assigned in the configuration using a similar syntax as NED pattern assignments:

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

One can also write expressions, including stochastic expressions, in NED files and in ini files as well:

```
**.sendInterval = 1s + normal(0s, 0.001s)
```

NED files (together with C++ code) are considered to be part of *the model*, and to be more or less constant. Thus, parameters that are expected to change (or make sense to be changed) during *experimentation* should be put into ini files. A non-default value assigned from NED cannot be overwritten later in NED or ini files.

The NED Language: Parameters

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).

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

The NED Language: Parameters

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 numbers drawn from a random number generator). In contrast, non-volatile parameters are evaluated only once. An example:

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

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.

The NED Language: Parameters

Thus, if `serviceTime` is assigned an expression like `uniform(0.5s, 1.5s)`, every job will have a different, random service time.

Another configuration:

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

Volatile parameters are typically used as a configurable source of random numbers for modules. (A non-volatile parameter can be assigned a random value but the simulation would use a constant value chosen randomly at the beginning of the simulation.)

The NED Language: Parameters

Units

One can declare a parameter to have an associated unit of measurement, by adding the `@unit` property. The OMNeT++ runtime does a full and rigorous unit check on parameters to ensure *unit safety*. Constants should always include the unit.

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

Values assigned to parameters must have the same or compatible unit, i.e. `@unit(s)` accepts milliseconds, minutes, etc, and `@unit(byte)` accepts kilobytes, megabytes, etc, see Appendix.

The NED Language: Parameters

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 use XML syntax.

OMNeT++ contains built-in support for XML files. Using an XML parser, OMNeT++ reads and validates the XML file, caches the file, 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.

The NED Language: Gates

Gates are the connection points of modules. OMNeT++ has three types of gates: input, output and inout.

One can create single gates and gate vectors. The size of a gate vector can be given in the declaration, but it is also possible to leave it open. The ++ operator that automatically expands the gate vector. The gate size can be queried from various NED expressions with the `sizeof()` operator. See the example modules in the Quickstart section above.

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()` should be annotated with `@directIn`. It is also possible to turn off the connectivity check for all gates within a compound module (`allowunconnected` keyword).

The NED Language: Gates

```
simple Classifier {
  parameters:
    int numCategories;
  gates:
    input in;
    output out[numCategories];
}
simple Sink {
  gates:
    input in[];
}
simple TreeNode {
  gates:
    inout parent;
    inout children[];
}
simple BinaryTreeNode extends TreeNode {
  gates:
    children[2];  }
```

The NED Language: 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. A submodule is usually given statically, but it is also possible to specify the type with a string expression (parametric submodule types).

NED supports *submodule arrays* (vectors) and *conditional submodules* as well. Submodule 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.

A submodule may also have a curly brace block as body, where one can assign parameters, set the size of gate vectors, and add, or modify, properties like the display string. It is not possible to add new parameters and gates.

The NED Language: Submodules

```
module Node
{
  gates:
    inout port[];
  submodules:
    routing: Routing {
      parameters:   // this keyword is optional
        routingTable = "routingtable.txt"; // assign par.
      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:
    ...   }
```

The NED Language: Connections

Connections are defined in the connections section of compound modules. One can connect two submodule gates, or a submodule gate and the *inside* of a gate of the *parent* module. Connections cannot span across hierarchy levels.

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

The *gatename*++ notation causes the first unconnected gate index to be used. If the size of the submodule gate vector is open, the ++ operator expands the gate vector by one (the model structure is built in top-down order).

The NED Language: Connections

Channel specifications are similar to submodules. The following connections use channel types definde above. The code shows the syntax for assigning parameters and specifying a display string.

```
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++;
```

The default name given to channel objects is `channel`. It is possible to specify the name explicitly, and also to override the default name per channel type.

The NED Language: Connections

Connection parameters, similarly to submodule parameters, can also be assigned using pattern assignments, albeit the patterns are a little more complicated and less convenient to use. (Using bidirectional connections is a bit trickier.)

```
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;
}
```

Simulation Concepts: C++ Classes

Systems that can be viewed as discrete event systems can be modeled using *discrete event simulation*.

The time when events occur is often called *event timestamp*; with OMNeT++ we use the term *arrival time* (the word timestamp is reserved for an 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.

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.

Simulation Concepts: C++ Classes

```
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)
```

OMNeT++ uses *messages to represent events*. Each event is represented by an instance of the cMessage class or one its subclasses. *Simple modules* encapsulate C++ code that *generates events and reacts to events*, in other words, implements the behaviour of the model.

Simulation Concepts: C++ Classes

Messages are sent from one module to another. 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. If two arrival times are equal, the one with the smaller *scheduling priority* value is executed first.

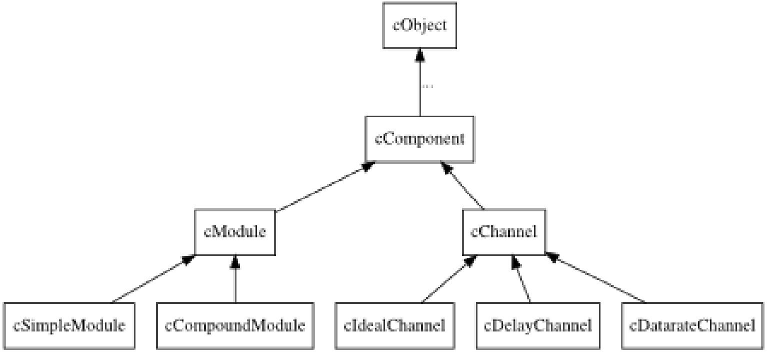
The current simulation time can be obtained with the `simTime()` function. Simulation time in OMNeT++ is represented by the C++ type `simtime_t`, by default a int64-based typedef to the SimTime class. The `dbl()` method of SimTime converts to double. Several functions and methods have overloaded variants that directly accept SimTime.

Simulation Concepts: C++ Classes

OMNeT++ simulation models are composed of modules and connections. Modules may be simple 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. Channels are also programmable in C++ by the user. Modules and channels are called *components*.

Components are represented with the C++ class cComponent. The abstract module class cModule and the abstract channel class cChannel both subclass from cComponent. cModule has two subclasses: cSimpleModule and cCompoundModule. The user defines simple module types by subclassing cSimpleModule. The cChannel's subclasses include the three built-in channel types: cIdealChannel, cDelayChannel and cDatarateChannel. The user can create new types by subclassing cChannel or other channel classes.

Simulation Concepts: C++ Classes



Signal-Based Statistics Recording

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,
- emitting information for other purposes.

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.

Signal-Based Statistics Recording

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. (The `registerSignal()` method takes a signal name as parameter, and returns the corresponding `simsignal_t` value; the `getSignalName()` method does the reverse.)

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. An example:

```
emit(lengthSignalId, queue.length());
```

The value can be of type `long`, `double`, `simtime_t`, `const char *`, or cObject *. Other types can be cast into one of these types, or wrapped into an object subclassed from cObject.

Signal-Based Statistics Recording

One use of signals is to expose variables for result collection without telling where, how, and whether to record them. Modules only publish the variables, and the actual result recording takes place in listeners. Listeners may be added by the simulation framework, or by dedicated modules. OMNeT++ approach goals:

- A controllable level of detail, you record all values as a time series, or only the mean, time average, minimum/maximum, standard deviation, etc, or the distribution as a histogram;
- Processing the results before recording them, for example to record the percentage of time the value is nonzero or over a threshold, record the sum of the values, etc;
- Aggregate statistics, e.g. the total number of packet drops or the average end-to-end delay for the whole network;
- Combined statistics, for example a drop percentage;
- Ignoring results generated during transient periods, etc.

Signal-Based Statistics Recording

Declaring Statistics

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 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). One can also specify a descriptive name for the statistic, and also a measurement unit.

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.

Signal-Based Statistics Recording

A queue length statistic, represented with an indexed property:

```
simple Queue
{
  parameters:
    @statistic[queueLength](record=max,timeavg,vector?);
  gates:
    input in;
    output out;
}
```

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. The maximum and the time average values will be recorded as scalars. One can instruct the simulation to record all results; this will turn on optional record items, those marked with a question mark, and then the queue lengths will be recorded into an output vector. The configuration lets you fine-tune the list of result items.

Signal-Based Statistics Recording

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.

```
simple Queue
{
  parameters:
    @signal[qlen](type=int); // optional
    @statistic[queueLength](source=qlen;
      record=max,timeavg,vector?);
    ...
}
```

This simple module type assumes that the module's C++ code emits a `qlen` signal, and declares a `queueLength` statistic based on that. (Declaring signals is currently optional and in fact `@signal` properties are ignored by the system.)

Signal-Based Statistics Recording

See the User Manual for the list of **Property Keys**, and the list of **Available Filters and Recorders**.

Naming of Recorded Results: ⟨*statisticName*⟩ : ⟨*recordingMode*⟩

```
@statistic[dropRate](source="count(drop)/count(pk)";
  record=last,vector?);
@statistic[droppedBytes](source="packetBytes(pkdrop)";
  record=sum,"vector(sum)?");
```

(The statistics sources combine multiple signals; there are quotation marks because the NED language parser does not allow parentheses in property values.)

These statistics will produce the following scalars:
`dropRate:last`, `droppedBytes:sum`
and the following vectors:
`dropRate:vector`, `droppedBytes:vector(sum)`

Configuring Signal-Based Statistics Recording

Recording can be tuned with the `result-recording-modes` configuration option. The statistic is identified by the full path (hierarchical name) of the module or connection channel object in question, plus the name of the statistic.

This configuration option accepts one or more items as value. An item may be a result recording mode, and two words with a special meaning, `default` (the set of non-optional items from the statistic record list) and `all` (all items from the list).

For example, if the following statistic is declared:

```
@statistic[foo](record=count,mean,max?,vector?);
```

We can write the following ini file lines (results are in comments).

Configuring Signal-Based Statistics Recording

```
**.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 with a more specific option key. The following ini line applies to queueLength statistics of `fifo[]` submodule vectors anywhere in the network.

```
**.fifo[*].queueLength.result-recording-modes = +vector
                    # default modes plus vector
```