Omnet++

https://doc.omnetpp.org/omnetpp/manual/#fig:ch-overview:modules

Overview

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.

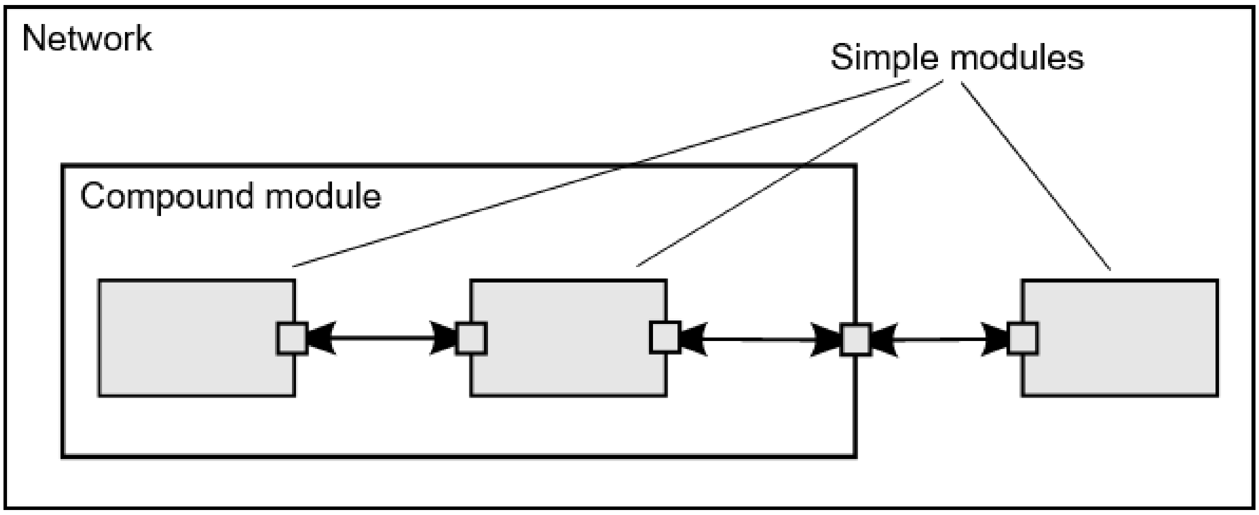


Figure: Simple and compound modules

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

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.

Omnet models

- `.ned` – topology descriptions
- `.msg` – message definitions
- `.h/.cc` – module sources. C++ files

NED

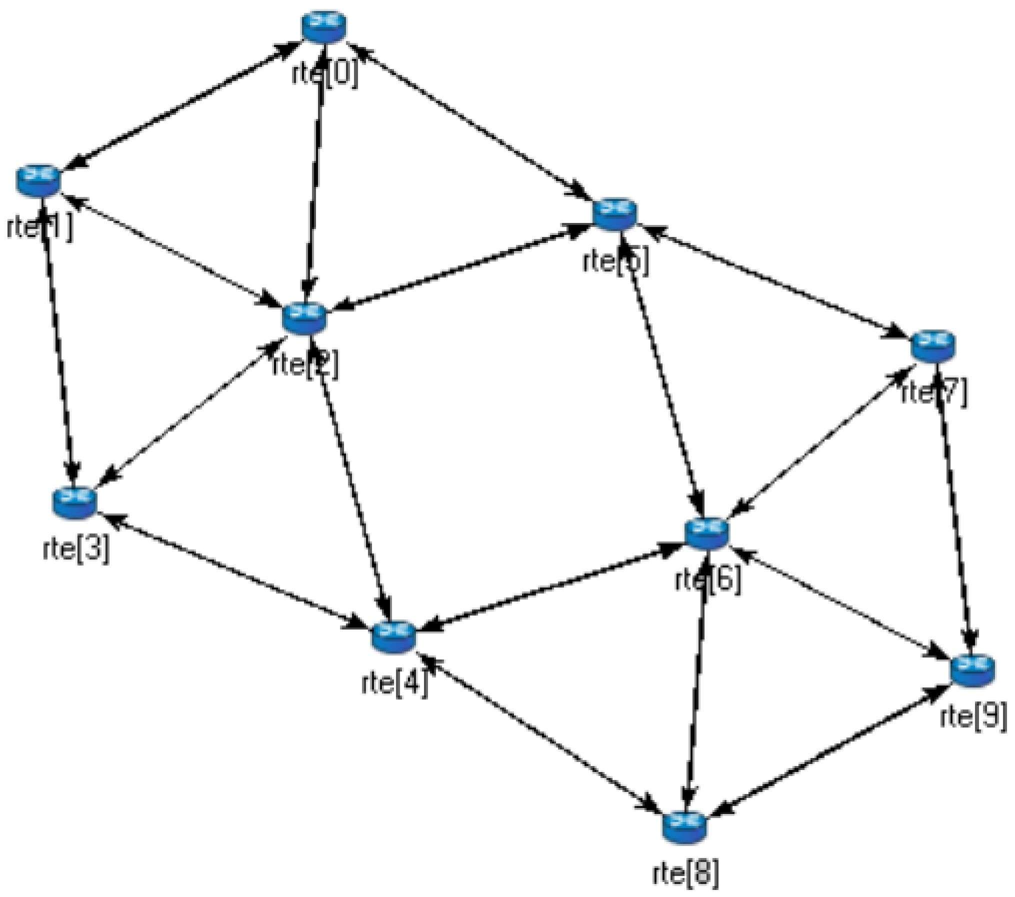


Figure: The network

Quickstart

Network

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

Channel

```
//
// A Network
//
network Network
{
    // New channel type
    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++;
        ...
}
```

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).

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.

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

simple Routing {
    ...
}

simple Queue {
    ...
}
```

Compound modules

```
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];
        }
}
```

Simple modules

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

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

subclassing

Specialization – set some parameters

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

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

Subclassing

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


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

Compound modules

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.

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

Channels

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

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

// Predefined ned classes
// ned.IdealChannel, ned.DelayChannel, ned.DatarateChannel
channel Ethernet100 extends ned.DatarateChannel {
    datarate = 100Mbps;
    delay = 100us;
    ber = 1e-10;
}
```

Parameters

more examples

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

Parameters can be of type `double, int, bool, string, xml`; they can also be declared `volatile`

Parameters can get their value from NED files or 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.

Gates

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

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

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

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.

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.

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

Connections

```
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;
}
```

Channel names with `name: type.`

Common patterns

Simple modules

Simulation concepts

pretty clear explanations here

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.

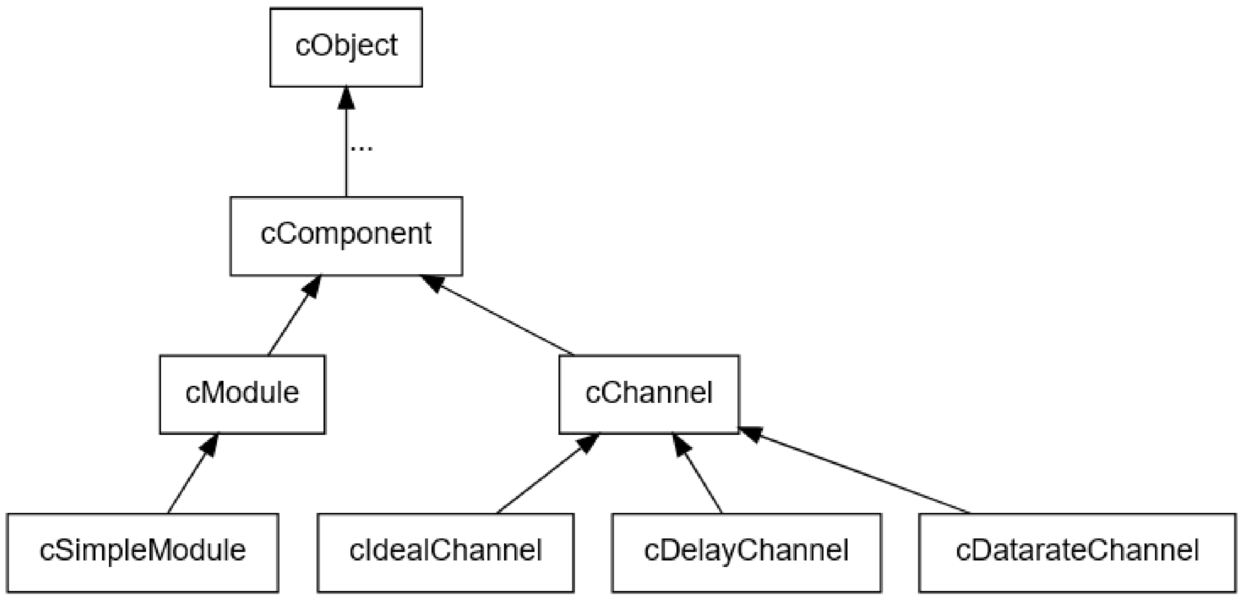
- OMNeT++ uses **messages** to represent events.
- Messages are represented by instances of the `cMessage` class and its subclasses.
- Messages are sent from one module to another – this means that the place where the "event will occur" is the message's destination module, and the model time when the event occurs is the arrival time of the message.
-

Events are consumed in **arrival time order**
Scheduling priority is a user-assigned integer attribute of messages.

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

Components, Simple modules, channels

Inheritance Hierarchy



Setup and cleanup

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

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

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

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

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

`constructor()`

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

`initialize()`

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

`finish()`

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

`destructor()`

- Delete everything which was allocated by new and is still held by the module class. With self-messages (timers), use the cancelAndDelete(msg) function

Functionality

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

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

The `receive()` and `wait()` functions cannot be used in `handleMessage()`

Example of a Generator that sends messages

```
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);
}
```

About `activity` – used like coroutines / multithreading (Extra read)

Parameters

Accessed with `par`

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

Convert to desired type with `boolValue(), longValue(), doubleValue(), stringValue(), stdstringValue(), xmlValue()`
Set the parameter values with `setBoolValue(), setLongValue(), setStringValue(), setDoubleValue(), setXMLValue()`
Extra methods: `getName(), getType(), getTypeName()`

Note

- non-volatile NED parameters are constants – they are read once
- 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.

Gates

```
// For in / out gates
cGate *outGate = gate("out");
// for inout gates use $i or $o
cGate *gIn = gate("g$i");
cGate *gOut = gate("g$o");
// or
cGate *gIn = gateHalf("g", cGate::INPUT);
cGate *gOut = gateHalf("g", cGate::OUTPUT);
```

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

Pointers to previous and next gates with `getPreviousGate() and getNextGate()`
Extra methods: `getName(), getType()`, for inout gates the name contains `$i , $o` so use `getBaseName(), getNameSuffix()`

```
// Get channels with
cChannel *channel = gate->getChannel();
```

```
// Get back the gates with
cGate *gate = channel->getSourceGate();
// They may result in nullptr if the channels /gates are not found
```

Sending and recieving messages

Scheduling events

```
// Message to itself to schedule events
scheduleAt(absoluteTime, msg);
scheduleAt(simTime()+delta, msg);

// Handling the message will be done in `handleMessage()`.
msg.isSelfMessage(); // To check for self messages
cancelEvent(msg);`  // Cancel self messages

cancelAndDelete();// uesd in destructors
```

```
// To reschedule use this pattern:
if (msg->isScheduled())
    cancelEvent(msg);
scheduleAt(simTime() + delay, msg);
```

Sending messages

```
send(cMessage *msg, const char *gateName, int index=0); // Send by name
send(cMessage *msg, int gateId); // Send by id
send(cMessage *msg, cGate *gate); // Send by pointer

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

IMPORTANT

- In broadcast / retransmission **do not** reuse the message with multiple `send()` operations. **Duplicate it**.

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

// or shorter
int outGateBaseId = gateBaseId("out");
```

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

Delayed sending

- Better than `scheduleAt() + send()`

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

Send directly to an input gate

```
sendDirect(cMessage *msg, cModule *mod, int gateId);
sendDirect(cMessage *msg, cModule *mod, const char *gateName, int index=-1);
sendDirect(cMessage *msg, cGate *gate);
// Ex:
cModule *targetModule = getParentModule()->getSubmodule("node2");
sendDirect(new cMessage("msg"), targetModule, "in");
```

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

Module hierarchies

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.

`cComponent` ids can be recovered with `.getId()`

```

int componentId = getId();
```

```
// Getting information from parents
cModule *parent = getParentModule();
double timeout = getParentModule()->par("timeout");
```

```
// Getting info from submodules
int submodID = module->findSubmodule("foo", 3); // look up "foo[3]"
cModule *submod = module->getSubmodule("foo", 3);
```

Extra patterns

Signals

Signals are emitted by components (modules and channels).

Used for:

1. exposing statistical properties of the model, without specifying whether and how to record them
2. receiving notifications about simulation model changes at runtime, and acting upon them
3. **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
4. emitting information for other purposes, for example as input for custom animation effects

Properties

- Signals **propagate** on the module hierarchy up to the root. **At any level** one can register listeners, that is, objects with callback methods
- Signals are identified by signal names (i.e. `string`)
- Listeners can subscribe to signal names or IDs, regardless of their source.

How to use

```
// New signal
simsignal_t lengthSignalId = registerSignal("length");
const char *signalName = getSignalName(lengthSignalId); // --> "length"

// Emit signals: Takes a signal id and a value
// Value can be bool, long, double, simtime_t, const char *, or (const)
cObject *
emit(lengthSignalId, queue.length());
```

```
// Register a listener
cIListener *listener = ...;
simsignal_t lengthSignalId = registerSignal("length");
subscribe(lengthSignalId, listener); // By Id
subscribe("length", listener); // directly by name

// Unsubscribe
unsubscribe(lengthSignalId, listener); // or
unsubscribe("length", listener);
```

They can be doccumented in NED files

```
simple Queue{
    parameters:
        @signal[queueLength](type=long);
```

```
                ...
}
```