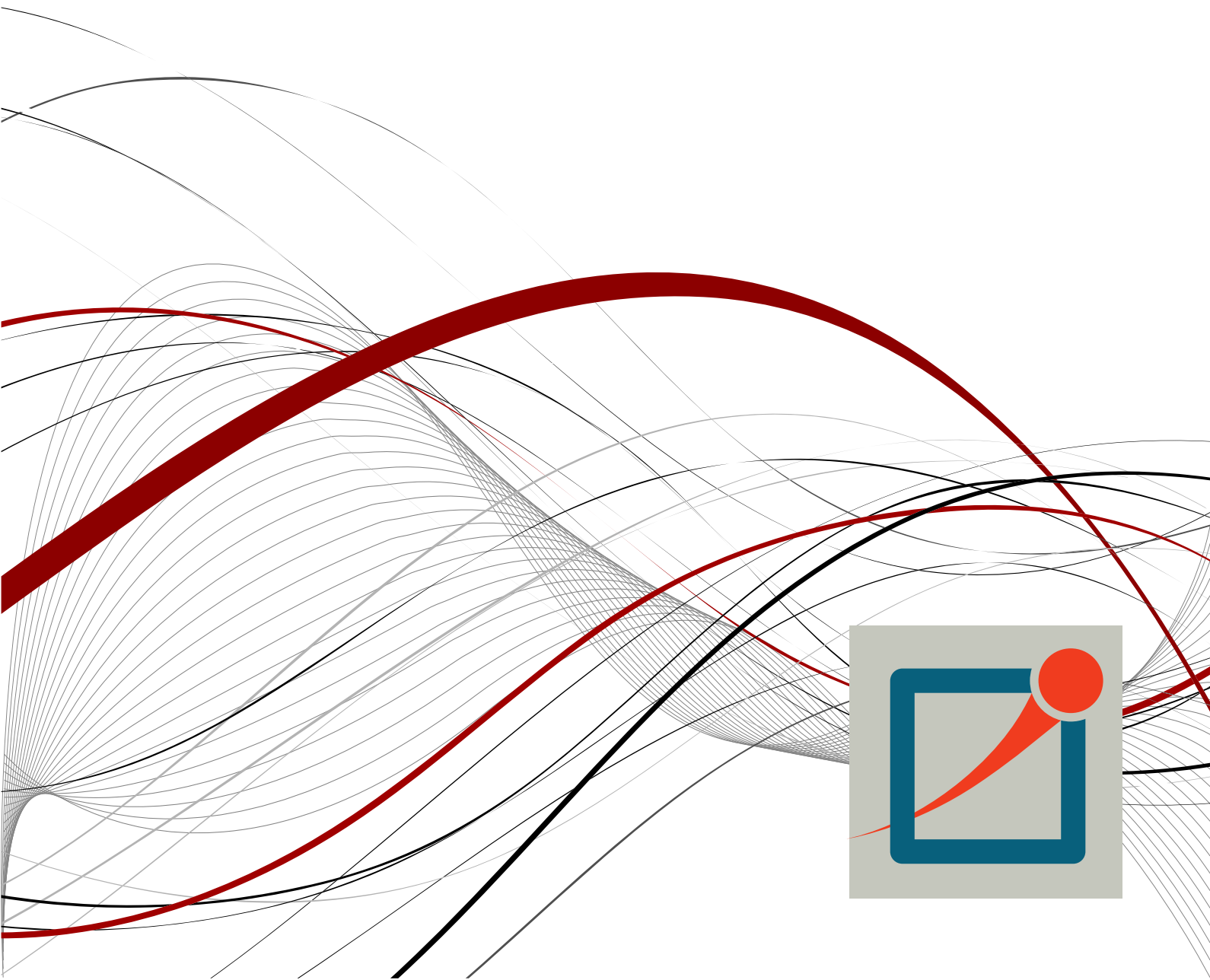
OMNeT++

Migration Guide

Version 4.4



Copyright © 2011 András Varga and OpenSim Ltd.

Chapter 1. What has changed since 3.x?

Overview

Simulation models written for OMNeT++ 3.x cannot be used directly with OMNeT++ 4.0 or later, due to changes in the C++ API, NED, ini and msg files. This document describes how to convert a 3.x model to run under OMNeT++ 4.x.

You should be already familiar with the OMNeT++ 3.x and 4.x before doing the migration. We recommend to take a closer look at the 4.x sample simulations before proceeding.

NED files

The NED language was significantly revised, and in addition to changing to a more consistent syntax, it was also expanded with new powerful new concepts: inheritance, module and channel interfaces, inner types, bidirectional connections, package structure, metadata annotation (properties), and so on. The following bullets list the changes that are important when porting models from the 3.x release.

- Curly braces has been introduced at the following places: module and channel definitions (the `endsimple`, `endmodule`, `endnetwork`, `endchannel` keywords have been removed); in submodules; around channel parameters in connections.
- The syntax of parameter and gate declarations has changed from Pascal style to C style.
- The `numeric` parameter type no longer exists, and must be replaced with `int` or `double`, depending on the parameter usage.
- The `const` keyword has been removed, and a new keyword `volatile` has been introduced. In 3.x an unqualified parameter was `volatile`; in 4.x it is `const`.0.
- The display string has been turned into a property with the `@display(...)` syntax.
- The `input` keyword was removed, and parameter prompt string has also become a property: `@prompt(...)`.
- Introduced a new parameter property, `@unit(...)`, to specify physical units. For parameters with units, all values in ini and NED files must be given with the same or a convertible unit, otherwise an error will be signaled.
- The `ref` keyword was removed, because parameters are now always passed by reference.
- `ancestor` parameters have been removed.
- The `gatesizes` section in compound modules has been renamed to `gates`.
- Conditional parameter and gatesize sections are no longer supported. In most cases, they can be substituted using the `?:` operator.
- `connections nocheck` is now called `connections allowunconnected`.
- The syntax of the connection `for` loop has changed
- There is no more implicit conversion between `bool` and `long`/`double`.

- The `import` declarations now refer to fully qualified package or type names instead of files.
- You can place submodules directly into a network instead, of creating a compound module and separately declaring it to be a network as was required in 3.x.

Message (msg) files

- The field property syntax has been changed to be same as for NED files.

Initialization (ini) files

- The `[Cmdenv]`, `[Tkenv]`, `[Parameters]`, `[Partitioning]`, `[OutVectors]` sections no longer exist, and their contents should be copied under the `[General]` section.
- Configuration options from `[Cmdenv]` and `[Tkenv]` have been prefixed with `cmdenv-` and `tkenv-`, respectively.
- The `[Run 1]`, `[Run 2]`, ... sections are no longer used and should be converted to named configurations: `[Config First]`, `[Config Second]`, etc. Note that run numbers no longer refer to configuration sections but to iteration numbers.
- The `cmdenv-express-mode` option (which was `express-mode` under `[Cmdenv]`) defaults to `true` instead of `false`.
- Most options in the `[Tkenv]` section have been removed, except the following ones: `tkenv-default-run`, `tkenv-image-path`, `tkenv-plugin-path`.
- The `tkenv-default-run` option (which was `default-run` under `[Tkenv]`) used to refer to a section. Now it refers to an iteration number, so now it only makes sense together with the new `tkenv-default-section`.
- There is a new `cmdenv-interactive` option defaulting to `false`, which causes Cmdenv to never read the standard input, and abort on missing parameter values. In 3.x, the default behavior was to read values from stdin.
- The `preload-ned-files` option has been removed, because in 4.x, NED files are loaded from directories in the NED path. (The NED path is a string that contains a list of directories, and it it may come from the `NEDPATH` environment variable, from a command-line option, or from the `ned-path` ini file option. For single-directory simulation models, the default value '.' should be sufficient.
- In 3.0 the `network` option was referring to a NED type loaded from one of the files specified in the `preload-ned-files` option. In 4.x, it specifies a qualified name referring to a NED type, which must be available under the directories specified in the `ned-path` option. In most cases, the `network` option will work unmodified.
- The `**.somepar.use-default=true` syntax should be changed to `**.somepar=default`. `**=default` does not need to be written out, because it is the default.
- Several configuration options have been renamed or otherwise changed. For further details, see `src/envir/ChangeLog` and other `ChangeLog` files.

Makefiles

The makefile generation and the make process has been rewritten. Notably, a single **opp_makemake --deep** command may replace complicated makefile systems for

multi-directory models like the INET Framework. Check the **opp_makemake -h** for further information.

The makefile generator can generate three types of makefiles:

- Local (default): only the sources from the current directory will be included, subdirectories will be ignored. This mode is recommended for single directory projects.
- Recursive (`--recurse`): includes files from the current directory and calls into all subdirectory. Makefiles should exist in all subdirectories.
- Deep (`--deep`): this is the preferred mode for multi-directory projects. All source files are automatically gathered from all subdirectories recursively. Only a single makefile is generated in the root directory. Include path for the project is automatically discovered. Directories can be excluded with the `-X` option.

C++ code (cc/h files)

This is just a brief summary of API changes since 3.x. Please consult the `include/ChangeLog` file for detailed information.

- Several header files have been renamed in `omnetpp/include`. (This should not affect simulation models, as they should only include `<omnetpp.h>`.)
- Renamed: `cObject` became `cOwnedObject`, `cPolymorphic` became `cObject`, and `cNamedObject` was introduced in between. Several method names have changed in different classes. Check the `ChangeLog` file for details.
- Added the `get` verb to the names of nearly all getter methods
- `<omnetpp.h>` now provides the C99 integer types and limit macros, even on systems that don't have `<stdint.h>`
- `simtime_t` is now not double but class `SimTime` (64-bit fixed point number)
- Added simtime compatibility mode: If needed, the simkernel can be compiled with `simtime_t = double`. For that, compile everything with `USE_DOUBLE_SIMTIME` defined (add `-DUSE_DOUBLE_SIMTIME` to `CFLAGS`).
- Introduced inout gates. Note: with inout gates, `gate("gatename")` does not work, use `gate("gatename$i")` or `gate("gatename$o")` instead
- Channels became first-class citizens: they have a common base class (`cComponent`) with `cModule`, they participate in the `initialize()`/`finish()` protocol, and so on
- Introduced `cComponent`, a common base class for `cModule` and `cChannel`. Some new methods to mention: `isModule()`, `getNedTypeName()`
- `cBasicChannel` renamed to `cDatarateChannel` and added `cIdealChannel` which lets messages through without any change and without any delay.
- Exception handling changed: now all our exceptions subclass from `std::exception` (e.g. `cException` extends `std::exception`), and exceptions are now thrown by value not by pointer.
- `cOutVector`: removed half-hearted tuple=2 support from `cOutVector` and underlying infrastructure and added methods for metadata annotation: `setEnum()`, `setUnit()`, `setType()`, `setInterpolationMode()`, `setMin()`, `setMax()`
- Numerous changes related to `cDisplayString`. Please check `include/ChangeLog` for details.

- cQueue: head()/tail() removed, back()/front() added (insertion at back, pop from front); because of the head/tail change, iterator direction and meaning of insertBefore()/insertAfter() has also changed. Additionally, the boolean flag to specify ascending/descending order has been removed.
- cMessage changes: the length, bit error flag and encapsulated message cMessage fields got factored out from cMessage, into the cPacket class (which extends cMessage). All network packets (frames, datagrams, etc) are now supposed subclass from cPacket, not directly cMessage.
- A new cPacketQueue class has been introduced to store cPackets and subclasses.
- Global findXXX(const char *name) functions turned into static cXXX::find(const char *name); methods (affected: findLink(), findFunction, findEnum, findChannelType, findNetworkType(), findModuleType())
- Run number handling were made entirely the matter of cEnvir. cSimulation::runNumber() got removed. Also removed run number from the arg list of cEnvir callback functions.

Environment variables

- The OMNETPP_BITMAP_PATH environment variable has been renamed to OMNETPP_IMAGE_PATH. The system will check this at runtime, and print a warning if the old variable is still present.

Command line options

- -f is now optional when specifying an ini file
- -r now refers to a run number instead of a named configuration in the ini file; usually -r is only meaningful with -c (which selects the configuration)
- Use the -h all switch to get detailed info about you simulation executable (or opp_run -h all to get info about OMNeT++ itself.)
- For further information use opp_run -h or the -h switch on any simulation executable.

Chapter 2. Migration tools

There are steps during the migration that can be easily automated. OMNeT++ 4 provides several command line tools that may help during the migration process. These tools are available under the migrate directory in the OMNeT++ 4.x installation.

migratened

The tool recursively migrates all `.ned` files under the current directory by doing the following:

- Converts all simple, module, network, channel type declarations to use the new curly brace format.
- Converts all parameter definitions to the new syntax.
- Removes the `const` qualifiers, and adds the `volatile` qualifier to non-`const` parameter definitions.
- For safety reasons, the automatic migration converts `numeric` parameters to `double`. Later the parameters must be manually checked if the type `int` would be sufficient, and change accordingly.
- Converts all submodule declarations to use the new curly brace syntax.

migratemsg

The tool recursively migrates all `.msg` files under the current directory by doing the following:

- Converts all properties to the new format.

migrateini

The tool recursively migrates all `.ini` files under the current directory by doing the following:

- Copies the entries from the `[Parameters]`, `[Cmdenv]`, `[Tkenv]`, `[OutVectors]` and `[Partitioning]` sections to the `[General]` section.
- Merges the entries from multiple occurrences of the `[General]` section into one.
- Prefixes the entries in the `[Cmdenv]` and `[Tkenv]` sections with `cmdenv-` and `tkenv-`, respectively (unless the entry already begins with that).
- Renames the `[Run 1]`, `[Run 2]`, etc. sections to `[Config config1]`, `[Config config2]`, etc.
- Renames all configuration entries that have changed.
- Changes `**.use-default` to `**=default`.

migratecpp

The tool recursively migrates all `.cc` and `.h` files under the current directory by doing the following:

- Renames all changed classes and methods that can be unambiguously identified in the code.

- Removes occurrences of obsolete macros (`Define_Module_Members()`, etc).
- Print warnings for all places that may need further inspection or manual changes.

opp_makemake

This tool is not a migration tool, but rather you will be able to create new makefiles for your project. Old makefiles cannot be reused.

Chapter 3. How to migrate

We recommend to port your simulation model in several stages:

1. Get it working with 4.x as fast as possible
   a. Run the automatic migration scripts.
   b. Do manual changes to your model and use as few of the new features as possible.
   c. Verify whether your model is working correctly and produces the same results as the old one. (either exactly or statistically)
2. Improve it by making use of new OMNeT++ features.

Getting your simulation model working

1. PREREQUISITES: Have OMNeT++ 4.x installed and working, and familiarize yourself with the IDE.
2. Make a backup of your simulation model. Be prepared to start over with the migration several times, until you get it right.
3. Change into the directory of your simulation model, and run all scripts in the `<omnetpp>/migrate` subdirectory from there.

   ```
   $ cd MyModel
   $ ../omnetpp-4.1/migrate/migratened
   $ ../omnetpp-4.1/migrate/migrateini
   $ ../omnetpp-4.1/migrate/migratemsg
   $ ../omnetpp-4.1/migrate/migratecpp | tee migratecpp.out
   ```

   The scripts convert NED, ini, msg and C++ files to 4.x format. The result will need some manual post-processing, because not everything can be converted automatically. The scripts will print some hints on what you'll need to do manually -- please make note of these printouts. Especially, `migratecpp` is going to print a number of notes, warnings and hints -- read them carefully.
4. If your simulation model is based on the INET Framework, have the new INET installed, and similarly run the scripts in the `migrate/` subdirectory of INET. They will update your source files according to changes in the INET Framework.

   ```
   $ cd MyModel
   $ ../INET/migrate/migratened
   $ ../INET/migrate/migrateini
   $ ../INET/migrate/migratemsg
   $ ../INET/migrate/migratecpp | tee migratecpp.out
   ```
5. You can do the rest of the migration either on the command line, or in the OMNeT++ IDE. We recommend the latter. To use the IDE, you need to create a project for your simulation model. Select **File | New | OMNeT++ Project...** from the menu. A wizard comes up. On the first page, uncheck "Use default location" and specify the name and the directory of your simulation model, then go through the other wizard pages and hit Finish at the end. You should see the new project appear in the Project Explorer (left), and it should contain your files. If something goes wrong, remove the project by selecting it and hitting DEL. It will ask whether you also want to delete the files from the disk -- answer "no". Then start over with project creation.
6. If your project is based on INET (or any other project), you can set up the project as one that depends on the INET project. To do that, make sure the INET project

is imported and open, then open the Properties dialog for your project (select the project, right-click it, then choose Properties from the context menu), and check INET on the Project References page. This makes the NED types of INET available in your project, and also puts INET directories on the C++ include path. Make sure the INET project builds a (static/shared) library, not an executable, so your project can link with it -- you can check that by opening INET's Project Properties dialog, and going to the C/C++ Build / Makemake page.

7. NED in 4.x has a package system, similar to Java. If your model contains NED files in several subdirectories, these subdirectories now mean packages, and the NED files will need package declarations and imports to be added. This can be done automatically in the IDE. Have your project created and open in the IDE (see previous step), then choose **Project | Clean up NED files...** from the menu. Select your project and click OK. The IDE will then fix all package declarations and imports in your NED files.

   You may want to add a `package.ned` file to define the root package -- this will be described in a later section.

8. Revise NED files. This includes:

   - revise `volatile` parameters if they really need to be volatile

     Some superfluous `volatile` qualifiers might pop up for parameters where the original model did not specify `const`. It is safe to delete the `volatile` qualifier from parameters which are expected to be constant over the simulation. As a rule of thumb, a parameter needs to be volatile if it is being read during simulation, not only in the initialization phase. If it is only read from `initialize()`, remove the `volatile` keyword.

   - revise `double` parameters whether they should rather be `int`

     The `numeric` parameter type from the 3.x version is automatically converted to `double`, but you may need to change it to `int` if needed. Be sure to change the corresponding code in your C++ files as well.

   - `somepar = input;` lines became just `somepar;` -- you probably want to remove them

     The `input` keyword is no longer supported in NED files, but you can specify the value for this parameter in the .ini file as `**.somepar=ask`, which has the same effect.

   - remove superfluous network declarations

     A 3.x-style network declaration denotes a compound module as network. In 4.x, a compound module may be directly declared to be a network, so the extra step is not needed. Example: the 3.x network declaration

     ```
     network cqn : CQN
     endmodule
     ```

     is converted by the migration script into an inheritance:

     ```
     network cqn extends CQN {
     }
     ```

     However, you can remove that altogether, if you change the CQN module's declaration to use the `network` keyword (like: `network CQN {...}`), and replace `network=cqn` with `network=CQN` in the ini files.

- "like" module types should be changed into interfaces, and actual types declared to be "like" them

  For example, if you have a submodule

  ```
  app: <appType> like App;
  ```

  then App should be turned into a module interface (and its name prepended with "I" to conform to naming conventions), like this:

  ```
  moduleinterface IApp {
      gates:
          input in;
          output out;
  }
  ```

  and the concrete types should be modified to comply with IApp:

  ```
  simple BurstyApp like IApp { ... }
  simple AnotherApp like IApp { ... }
  ```

9. Compile your simulation model (right-click on the project and select Build from the context menu, or close all other projects and hit Ctrl+B.) The most frequent compile errors and their fixes:

   - "Cannot convert SimTime to double"

     simtime_t now maps to the int64-based SimTime class and not double. Wherever a simtime_t is assigned to a variable of type double, consider changing that variable to simtime_t as well. The new SimTime class does not provide implicit conversion to double because it would cause C++ ambiguity errors. Check the output of the migratecpp tool, as it gives you some hints what variables should be changed.

     Where still needed, use SIMTIME_DBL(t) to convert a simtime_t to double. In printf's, use "%s" and SIMTIME_STR(t). The advantage of using these macros instead of SimTime methods is that your model will also compile in -DUSE_DOUBLE_SIMTIME compatibility mode (see below).

     If your model is using double's extensively for time-related variables and you want to make a quickly and dirty port, OMNeT++ can be compiled with the original behavior, by specifying -DUSE_DOUBLE_SIMTIME in CFLAGS. However, be aware that you have to recompile all OMNeT++ libraries with this flag. We recommend to use the new SimTime type whenever possible.

   - "No such method setBitLength/getBitLength/encapsulate/decapsulate"

     Length and encapsulation have been moved to cPacket, a subclass of cMessage. You likely need to change the message keyword in .msg files to packet, which will cause the generated class to have cPacket as base class.

     ```
     message ABCPacket {...} ==> packet ABCPacket {...}
     ```

     Inside handleMessage() and other functions, cast the cMessage* pointer to cPacket*:

     ```
     cPacket *pkt = check_and_cast<cPacket *>(msg);
     ```

- "Cannot open file csimul.h" (or any other header)

  Only `<omnetpp.h>` is public API. Other OMNeT++ header files should not be included directly, as they may be renamed or removed in any future version.

- "sendDirect() does not take 3 (or 4) arguments"

  `sendDirect()`'s signature has changed. It used to take a delay as second argument; now it has two variants, one which takes no delay argument (i.e. if you have 0.0 in your simulation, just remove it), and another one that takes a propagation delay and a transmission duration. If you use the second one, you'll probably want to call `setDeliverOnReceptionStart(true)` on the receiver gate in the target module's `initialize()` method as well.

10Run your simulation model. The most frequent runtime errors and their fixes:

- "Cannot convert unit 'none' to 'seconds'"

  Physical units now have to be written out in expressions, so you need to change `5` into `5s`, and `exponential(1)` into `exponential(1s)`.

- "Cannot convert unit 'none' to 'bps'"

  The datarate channel parameter now has unit `bps` (bit/sec), and this unit must be written out. Kbps, Mbps, Gbps are also accepted.

- "No such module type 'X'"

  If your model creates modules dynamically, module types need to be looked up by fully qualified name (like `"some.package.X"`).

Making use of new OMNeT++ features

NED files

- Add default icons.

  It is now possible to give a default display string (containing an icon, etc) to module types; at runtime the default gets merged with the submodule display string to get the effective display string. To assign default icons to modules, you would move the `"i="` tags from submodule display strings into the corresponding simple module types. The result will be like this:

```
simple Node {
    @display("i=block/fork");
    ...
}

module Net {
    submodules:
        node1 : Node {
            @display("p=240,100"); // note: "i" tag moved out
        }
        ...
}
```

- Parameter values assigned in ini files could be put into the corresponding NED file as default values

If your ini file contains a lot of parameter values that usually do not change, consider to move the values to the NED file as parameter defaults. Use the following syntax:

```
int somepar = default(42);
```

- Use `@unit` for your module's parameters to specify physical units. This will enforce physical units in parameter values as well.

  ```
  volatile double interArrivalTime @unit(s);
  ```

- Make use of module inheritance.

  If you have several modules that share the same behavior but differ only in parameterization, you can take advantage of module inheritance. An example:

  ```
  simple Router {
      int ports;
  }

  simple Router8 extends Router { // still uses the "Router" C++ class!
      ports = 8;
  }

  simple Router16 extends Router {
      ports = 16;
  }
  ```

  Note: the C++ class will be inherited from the base module, that is, all three module types will use the `Router` C++ class, even if you have `Router8` etc classes in C++! To replace the C++ class as well, you need to add a `@class` property:

  ```
  simple AdvancedRouter extends Router {
      @class(AdvancedRouter);  // makes it use the "AdvancedRouter" C++ class
  }
  ```

  Inheritance can also be used to factor out the common part of several compound modules into a base type (because a derived compound module may add new submodules and connections, in addition to new parameters and gates).

- Use inout gates and bidirectional connections.

  A pair of uni-directional connections can be replaced with a single bidirectional one. The syntax:

  ```
      gates:
          inout port;
        ...
      connections:
          node1.port <--> node2.port;
  ```

- Use inner types

  If you use a type only locally, consider turning it into an inner type. This is especially useful with channels.

  ```
  module Network {
      types:
          channel Ethernet extends ned.DatarateChannel {
              datarate = 100Mbps;
          };
  ```

```
    ...
    connections:
        node1.port <--> Ethernet <--> node2.port;
        ...
}
```

- Define the root package for your NED files if you plan to give your model to other people. This can avoid name clashes with other models. To do that, put a `package.ned` file into the toplevel NED source folder (i.e. a directory that's listed on the OMNeT++/NED Source Folders page of the Project Properties dialog of your project). The package declaration in that `package.ned` file determines the package of that directory and all directories underneath. For example, if the file's contents is

  ```
  package org.myproject;
  ```

  then the NED packages will be `org.myproject`, `org.myproject.subdir1`, `org.myproject.subdir2`, etc.

- Use additional display string tags (module background, grid etc.) to enhance your simulation. See the manual for new supported tags.

- Possibly make use of `@properties` as "marker interfaces", like `@host(true)`

ini files

- Give meaningful names to configurations. In 3.x, only run numbers were allowed.

- Remove redundant parameter settings. If a module parameter now has a default value and the ini file explicitly sets that value, that line can be removed from the ini file.

If you open the file in the IDE, it will annotate the lines that can possibly be removed with a blue "i" mark.

- Make use of section inheritance, if applicable. Common settings may be factored out into a based config that other configs extend (syntax: add an `extend=BaseConfig` line). Example:

  ```
  [General]
  network = Aloha

  [Config SlottedAloha]
  **.slotted = true

  [Config SlottedAlohaLowTraffic]
  extends = SlottedAloha
  **.interval = exponential(1s)
  ```

- Make use of the iteration syntax (`${...}`).

  If you had many runs that simulated the same network with different parameter values, they can be merged into a single config that contains an iteration. For example, if you had

  ```
  [Run 1]
  **.numClients = 2

  [Run 2]
  **.numClients = 5
  ```

```
[Run 3]
**.numClients = 10
```

Then you can now merge them into a single config:

```
[Config MyExperiment]
**.numClients = ${N=2,5,10}
```

Another example:

```
**.iaTime = exponential(${mean=1,1.5,2,3,5..21 step 2}s)
```

If you have created external scripts previously to explore the result of different parameter combinations, you can also do it now without scripts. There can be several iterations within a section (nested loops), you can specify an additional constraint to get a subset of the Cartesian product (`constraint=`), and you can repeat each one several times with different seeds (`repeat=`).

C++ code

- You can add metadata to the output vectors to enhance the visualization. Use the `setInterpolationMode()`, `setEnum()` etc. methods of `cOutVector`.
- If possible take advantage of inout gates a bidirectional connections. use `gate("gatename$i")` or `gate("gatename$o")` to access the two direction separately (note: `gate("gatename")` will not work).
- Use `cMessage` or `cPacket` depending on your needs. The length, bit error flag and encapsulated message `cMessage` fields got factored out from `cMessage`, into the `cPacket` class.
- If you have a complex channel model, now you can extend `cChannel` and register your new class. Override the `deliver(...)` method. There are three new built-in channels you can use as a starting point: `cIdealChannel`, `cDelayChannel`, `cDatarateChannel`
- If a message is transmitted on a finite-datarate channel, call `setDeliverOnReceptionStart(true)` on a simple module's input gate to deliver the message to the receiver module at the start of the reception (instead of the default, which delivers at the end). In 3.x, the only way to receive the packet at the beginning of the reception was to set the channel datarate to zero and calculate the duration manually -- this workaround is no longer needed.
- In wireless models, the handling of transmission duration needs to be refactored.

  In 3.x, the receiver side usually calculated the frame duration independent of the sender. In 4.x, the duration calculated by the sender should be specified in the `sendDirect()` call, which writes it into the packet (`setDuration()`). The receiver module should have the input gate configured to deliver packets at the start of the reception, i.e. it is supposed to call `setDeliverOnReceptionStart(true)` on the gate in the initialize phase. The receiver module should obtain the duration from the packet (`pkt->getDuration()`) instead of recalculating it.
- You can use the `getProperties()` method of `cComponent` to access model metadata. Properties can be attached to a module or channel in a NED file with the `@propname(key1=val1,val2;key2=valA,valB)` syntax. You can use properties as a markers (e.g. `@host`), or to provide additional information about the model.