OMNET++ MANUEL

Information : ce document se base sur ce lien de documentation OMNeT++. Ce lien est référencé sous le nom de « document officiel » tout au long de ce texte. Cet écrit ne reprend pas le document dans son entièreté. Pour toute information manquante nécessaire à la compréhension d'OMNeT++, ainsi que pour tout erreur dans ce texte, n'hésitez pas à me contacter pour que puisse procéder aux modifications.

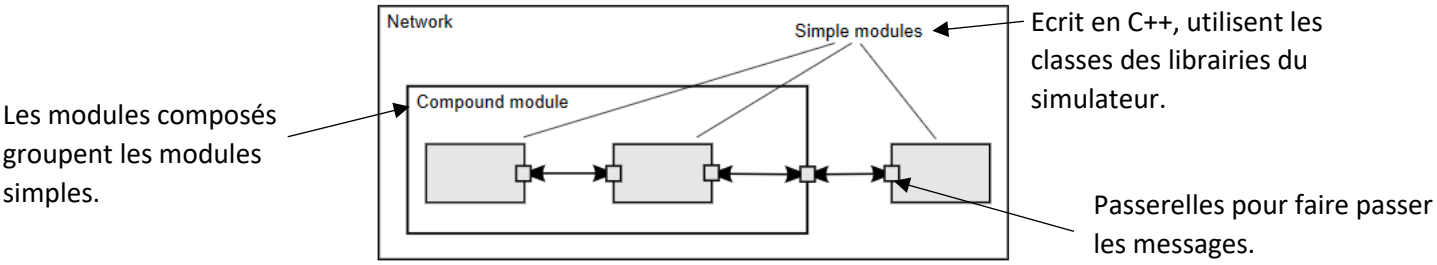
I. Présentation générale

OMNeT++ lui-même n'est pas un simulateur de quoi que ce soit de concret, mais fournit plutôt une infrastructure et des outils pour écrire des simulations.

Utilisé dans divers domaines :

- Modélisation des réseaux de communication filaires et sans fil
- Modélisation de protocole
- Modélisation des réseaux de file d'attente
- Modélisation de multiprocesseurs et autres systèmes matériels distribués
- …

Un modèle OMNeT++ consiste de modules qui communiquent avec des messages.



Le nombre de niveaux hiérarchiques/ d'emboitement des modules est illimité.

Un modèle OMNeT++ se compose de modules hiérarchiquement imbriqués qui communiquent en se transmettant des messages. Les modèles OMNeT++ sont souvent appelés réseaux. Le module de niveau supérieur est le module système. Le module système contient des sous-modules qui peuvent également contenir des sous-modules eux-mêmes.

Les objets de simulation (messages, modules, files d'attente, etc.) sont représentés par les classes C++.

1. Un simulateur DES

Système à événements discrets : système où l'état des événements change. Il est supposé que rien ne se produit entre deux événements consécutifs. Ces systèmes peuvent être modélisés à l'aide de la simulation d'événements discrets (DES). Cela implique qu'entre le début de la transmission d'un paquet et sa réception rien d'intéressant se produit.

Boucle événementielle : La DES maintient l'ensemble des futurs événements dans une structure de données appelée FES (Future Event Set) ou FEL (Future Event List). Ces simulateurs fonctionnent généralement selon le pseudo code suivant :

L'étape d'initialisation construit généralement les structures de données représentant le modèle de simulation.

Boucle qui consomme les événements du FES.

Les événements sont traités dans l'ordre d'horodatage pour être sûr qu'un événement actuel ne peut avoir un effet sur un événement antérieur. Si deux événements arrivent en même temps le message avec la priorité la plus haute est traité en premier.

```
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.)
```

Traitement indique des appels au code fournit par l'utilisateur.

La simulation se termine lorsqu'il n'y a plus d'événements ou quand il n'est plus nécessaire de faire jouer la simulation.

2. Envoi et réception de messages

Les messages sont un concept central dans OMNeT++. Les objets « message » représentent des événements, des paquets, des commandes, etc.

Les messages sont représentés par la classe `cMessage` et de sa sous classe `cPacket`. `cPacket` est utilisé pour les paquets réseau (trames, datagrammes, etc.) et `cMessage` est utilisé pour tout le reste. Des modules simples communiquent donc entre eux via la transmission de messages. Les modules simples créent, envoient, reçoivent, stockent, modifient, planifient, et détruisent les messages.

II. Langage NED

L'utilisateur décrit la structure d'un modèle de simulation dans le langage NED. NED signifie Network description. NED permet à l'utilisateur de déclarer les modules simples, ainsi que de les connecter et de les assembler en modules composés. Voici quelques une des fonctionnalités principales du langage NED :

- Hiérarchie : chaque module complexe peut être décomposé en modules plus simples et donc utilisé comme module composé.
- Basé sur des composants : les modules simples et composés sont « réutilisables ». Réduit la quantité de code.
- Héritage
- Paquet : la langage NED dispose d'une structure de paquets de type Java ce qui réduit les conflits de noms entre les différents modèles.

La langage NED peut être convertis en XML, et inversement, sans perte de données.

3. Introduction au langage

L'exemple suivant est tiré de la page officielle (cf. début du document pour accéder au lien) partie 3.2.

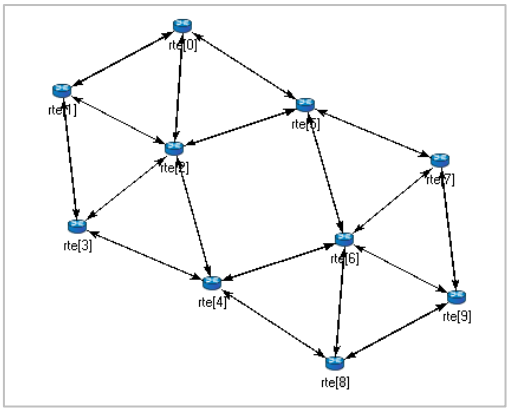


FIGURE 1 - RESEAU POUR L'EXEMPLE

Nom du réseau. Permet de dire au fichier INI quel réseau exécuter.

Commentaires :
- Code plus lisible
- Affichés dans les info-bulles...

```
//
// 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++;
        ...
}
```

Les nœuds sont connectés à un canal qui a un débit de 100 Mbps (Cf. 2.1.a).

Comment les nœuds doivent être reliés

Connexion bidirectionnelle

Ajoute un nouveau « gate », porte, au vecteur `port[]`.

FIGURE 2 - CODE NED REPRESENTANT LE RESEAU DE LA FIGURE 1

III. Différents types de modules

1. Modules simples

Les modules simples sont les composants actifs du modèle. Les événements se produisent à l'intérieur de ces modules. Un module simple n'est rien de plus qu'une classe C++ qui hérite de la classe `cSimpleModule`. Il faut enregistrer la classe auprès d'OMNeT++ avec la ligne « `Define_Module()` ».

Un module simple n'est jamais instancié par l'utilisateur. L'écriture du constructeur doit donc respecter des « normes » pour que le noyau de la simulation le comprenne :

- Pas d'argument
- En public

Exemple :

Méthode déclarée dans le cadre de `cComponent`. Appelée juste avant que la simulation commence à s'exécuter. Quand cette méthode n'est pas suffisante, une initialisation en plusieurs étapes est possible. Cf. partie 4.3.3.3 du document officiel.

```
// 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
}
```

`handleMessage()` est une fonction membre virtuel de `cSimpleModule` qui ne fait rien par défaut. L'utilisateur doit la redéfinir et ajouter le code de traitement de messages. `handleMessage()` est appelée pour chaque message qui arrive au module. Aucun temps de simulation ne s'écoule au cours d'un appel à cette fonction. Les modules avec cette fonction ne démarrent pas automatiquement. S'il est souhaité que le module fonctionne sans recevoir au préalable un message provenant d'autres modules il faut planifier les messages à partir de la fonction `initialize()`. En effet, le noyau crée uniquement des messages au démarrage pour les modules avec `activity()*`.

Les fonctions liées aux messages et événements qui peuvent être utilisées dans `handleMessage()` :

- Famille de fonction `send()` pour envoyer des messages à d'autres modules
- `scheduleAt()` : pour planifier un événement (le module s'envoie un événement à lui-même)
- `cancelEvent()` : pour supprimer un événement planifié.

Avantage de `handleMessage()`	*Inconvénient de `handleMessage()`*
- Consomme moins de mémoire - Rapide	- Les variables locales ne peuvent pas être utilisées pour stocker des informations d'état. - Besoin de redéfinir `initialize()`

`*activity()` : module simple qui peut être codé comme un processus ou thread. Lorsque la fonction `activity()` se ferme, le module est terminé. La simulation peut continuer s'il existe d'autres modules qui peuvent s'exécuter. Les fonctions les plus importantes qui peuvent être utilisées dans cette fonction sont :

- `receive()` : pour recevoir des messages.
- `wait()` : pour suspendre l'exécution pendant un certain temps.

- `send()`
- `scheduleAt()`
- `cancelEvant()`
- `end()`

Le problème avec la fonction `activity()` est qu'il n'évolue pas puisque chaque module a besoin d'une pile de coroutine distincte. Les coroutines sont similaires aux threads, mais sont planifiées non préventives. Pour plus d'informations sur cette fonction cf. partie 4.4.2 du document officiel.

`finish()` est une méthode déclarée dans le cadre de `cComponent`. Elle sert à enregistrer des statistiques. Appelée quand la simulation est terminée. Mais elle n'est pas toujours appelée, comme dans l'exemple présent au-dessus.

Il faut également déclarer le fichier NED associé. Les fichiers NED ne déclarent que les interfaces visibles du module (portes, paramètres).

Ici le fichier NED en lien avec l'exemple précédent serait :

```
// file: HelloModule.ned
simple HelloModule
{
    gates:
        input in;
}
```

Un autre *exemple :*

Mot clef qui désigne les modules simples. La première lettre du nom du module est en majuscule.

Chaîne d'affichage. Définit le rendu en mode graphique. Ici « `i=…` » définit l'icône par défaut.

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

simple Routing
{
    ...
}

simple Queue
{
    ...
}
```

Module simple pour la génération du trafic. Déclaration du module dans `App.ned`.

Module simple pour le routage. Déclaration du module dans `Routing.ned`.

Module simple pour la mise en attente. Déclaration du module dans `Queue.ned`.

Les sections `gates` et `parameters` sont facultatives. Il est possible de mettre des paramètres sans rajouter le nom de section « `parameters` » (Cf. partie 2.2 pour les paramètres).

Information : Les paramètres de module déclarés dans le fichier NED sont représentés avec la classe `cPar` au moment de l'exécution et sont accessibles en appelant la fonction membre ainsi : `par()` de `cComponent`.

2. Modules composés

Paramètres (tout en minuscule)

Permet une connexion bidirectionnelle.

Connexions entre les sous-modules. Cf. Partie 2.3.

Sous modules (déclarés dans la partie 2.1.b). En effet, Node (notre nœud) a besoin de ces sous-modules pour fonctionner.

```
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];
        }
}
```

Ici aussi toutes les sections (`parameters, submodules, gates, connections, types`) sont facultatives.

Les modules composés peuvent être étendu. L'héritage peut ajouter de nouveaux sous-modules, de nouvelles connexions, des paramètres, des portes… *Exemple :*

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

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

3. Sous-modules

Les sous-modules sont les modules contenus dans les modules composés.
Syntaxe de base :

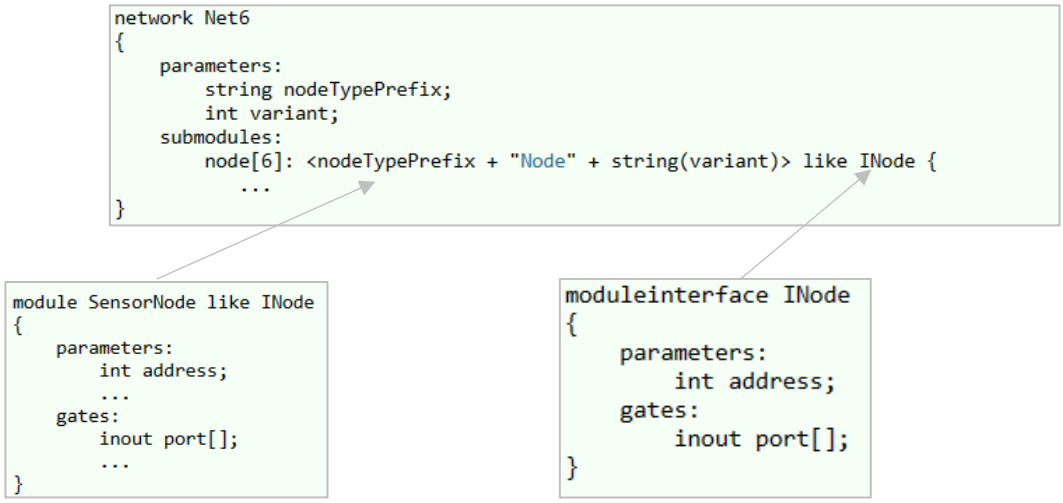
```
module Node
{
    submodules:
        routing: Routing;   // a submodule
        queue[sizeof(port)]: Queue;  // submodule vector
        ...
}
```

Un sous-module peut également avoir un bloc d'accolade comme corps. *Exemple :*

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

Sous-modules paramétriques : un type de sous-module peut être spécifié avec un paramètre de module de type `string`. Ce paramètre pourra prendre le nom de différents types, prédéfinis dans le code, en fonction des besoins. La syntaxe utilise le mot « `like` ».

Exemple :



IV. Zoom sur le fonctionnement d'OMNeT++

1. Les paramètres

Les paramètres sont des variables qui appartiennent à un module. Les types sont :

- `Double`
- `Int`
- `float`
- `bool`
- `string`
- `xml`
- `object`

Il est possible de donner l'unité de mesure avec : `@unit`.
Une valeur par défaut peut être donnée avec : `(default(…))`.

Exemples :

Pour faire correspondre n'importe quel index

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

FIGURE 3 - VECTEURS DE SOUS MODULES

Correspond à toute sous-chaîne ne contenant pas de point

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

FIGURE 4 - INDIVIDUELS (PAS VECTEURS)

** → correspond à n'importe quelle séquence de caractères (y compris les points).

Pour réattribuer la valeur par défaut à un paramètre : `nom_param = default`.

Pour demander à l'utilisateur à choisir la valeur du paramètre : **`nom_param = ask`**.

`x <=> y`	Compare deux éléments. Retourne : - 0 → x = y - Positifs (ex : 1) → x > y - Négatif (ex : -1) → x < y - `NAN` → y = `NAN`
`=~`	Comparer la chaîne de caractères. Renvoie un booléen. Sensible à la casse.
Les autres opérateurs sont décris partie 19.5.3 du document officiel.	

a. Paramètres volatils

Ce sont les paramètres marqués du mot clef **`volatile`**. Si un paramètre est marqué **`volatile`**, le code C++ qui implémente le module correspondant est censé relire le paramètre chaque fois qu'une nouvelle valeur est nécessaire, au lieu de le lire une fois et de mettre en cache la valeur dans une variable.

```
parameters:
    volatile double serviceTime;
```

b. Paramètres mutables

Ce sont les paramètres marqués du mot clef **`@mutable`**. Les paramètres mutables peuvent être définis sur une valeur différente pendant l'exécution, tandis que normal, c'est-à-dire que les paramètres non mutables ne peuvent pas être modifiés après.

c. Paramètres XML

Ce sont les paramètres marqués du mot clef **`xml`**. OMNeT++ prend en charge deux façons explicites de transmettre des données structurées à un module à l'aide de paramètres : paramètres XML et paramètres d'objet avec des données structurées de type JSON.

Exemple :

Déclaration d'un paramètre de type XML.

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

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

Accepte un nom de fichier comme argument ou permet de sélectionner un élément dans un code XML.

2. Portes

Les portes sont les points de connexion des modules. Il en existe 3 types :

- **`In`**
- **`Out`**
- **`Inout`**

Une porte ne peut être connectée qu'à une seule autre porte. Il est possible de créer des portes uniques ainsi que des vecteurs de portes. Il est souvent exigé que toutes les portes soient connectées. Pour ne pas vérifier la connectivité des portes il est possible de rajouter **`@loose`**.

Entrée pour recevoir des jobs qu'elle renvoie à une des sorties.

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

Nombre de sortie déterminée par le paramètre du module.

S'attend à recevoir des messages directement de l'émetteur.

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

Pour reconnecter une porte la balise **`@reconnect`** est nécessaire.

Exemple :

```
a.out --> {@reconnect;} --> b.in;
```

3. Connexions et canaux

Les connexions sont définies dans la section **`connections`** du module composé. Elles ne peuvent pas s'étendre sur plusieurs niveaux hiérarchiques.
Les portes sont spécifiées comme :

- ***`nom_sous_module.nom_porte`***: permet de connecter des sous-modules
- ***`nom_porte`*** : pour connecter les modules composés. La notation ***`nom_porte++`*** permet à la porte de premier indice non connectée d'être utilisée.

Les portes de modules sont représentées par des objets **`cGate`**. Il est possible de retrouver une porte par :

- Son nom : cependant cela ne fonctionne pas pour les portes **`inout`**.

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

- Son ID

Chemin : série de connexions (jointes avec des modules **`gates`**). Un chemin est dirigé. Il commence à une porte de sortie d'un module simple et se termine à une porte d'entrée d'un module simple. Ce chemin passe par plusieurs portes de modules composés.
Chaque objet **`cGate`** contient des pointeurs vers la porte précédente et la porte suivante dans la chemin. Un chemin peut être considéré comme une liste à double liaison.

Canal : l'objet canal associé à une connexion est accessible par Pointeur stocké à la porte source de la connexion. Les canaux encapsulent les paramètres et le comportement associés aux connexions. Les types de canaux sont comme de simples modules, dans le sens où ils sont déclarés dans NED.

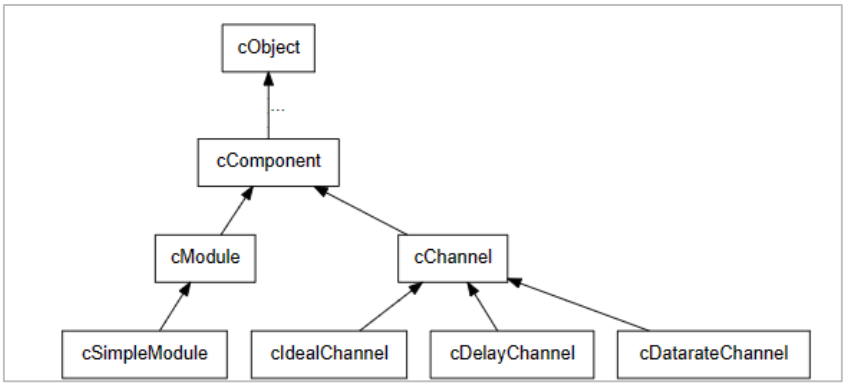
Pour éviter de répéter le débit de données pour chaque connexion il est possible de créer un nouveau type de canal :

```
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++;
    ...
```

Le nom par défaut donné à un objet canal est de base : ***`nom_canal : type`***. *Exemple :* `eth1 : EthernetChannel`.

Il existe de nombreux canaux prédéfinit. Il est donc rare de devoir en écrire. Les types prédéfinis sont :

- `ned. IdealChannel` : n'a pas de paramètres et laisse passer tous les messages sans retard ni effets secondaires. Une connexion sans objet canal est une connexion `IdealChannel`.
- `ned. DelayChannel` : a deux paramètres :
    - `delay` : délai de propagation du message. L'unité de temps doit être spécifiée. Toutes les unités de mesures sont disponible partie 19.5.11 du document officiel. *Exemple de spécification de l'unité* :
      ```
      volatile int packetLength @unit(byte) = default(4KiB);
      ```
    - `disabled` : booléen par défaut à faux. Quand il est a vrai l'objet Channel supprime tous les messages.
- `ned. DatarateChannel` : en plus des paramètres présents dans DelayChannel :
    - `datarate` : débit de données du canal. 0 → durée de transmission nulle, bande passante infinie. C'est la valeur par défaut.
    - `ber` : Bit Error Rate. Permet la modélisation des erreurs de base (réel entre 0 et 1).



Il est possible d'enlever « `ned.` » en rajoutant `Import NED.*`

Il est possible de rajouter des paramètres / de modifier des canaux existants via la sous-classification :

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

Les spécifications de canaux sont les flèches présentes dans la partie connexion. Parfois les noms des canaux peuvent être non spécifiés, le type sera retrouvé grâce aux paramètres rentrés.

Exemple :

Paramètre présent	*Type*
`Datarate`	`ned.DatarateChannel`
`Ber`	
`Per`	
Sinon les deux autres (cf. au-dessus)	

```
a.g++ <--> {delay=10ms;} <--> b.g++;
a.g++ <--> {delay=10ms; ber=1e-8;} <--> b.g++;
a.g++ <--> {@display("ls=red");} <--> b.g++;
```

Pour voir des exemples de connexions (arbre binaire, chaîne...) cf. la partie 3.10 du document officiel.

Il est également possible d'utiliser des types de connexions paramétriques. Cf. la partie 3.11.3 du document officiel.

4. Annotations des métadonnées

Il est possible de spécifier explicitement les classes C++ avec la propriété `@class`. Il est possible de mettre des classes avec des qualificateurs d'espace de nom comme `@class(mylib::Queue)` . Pour gagner du temps il est possible de mettre `@namespace(mylib)` dans le fichier `package.ned` pour avoir seulement à taper le nom de la classe, par *exemple* `Queue`.

Il existe d'autres métadonnées comme il a été spécifié dans les parties précédentes :

- `@display`
- `@prompt`
- `@loose`
- `@directIn`
- …

Pour avoir plusieurs propriétés avec le même nom, par *exemple* déclarer plusieurs statistiques produites par des modules, l'utilisation des `property indices` est possible. Il suffit de rajouter l'identifiant entre croquet après la balise.

Les propriétés peuvent également contenir des listes, des données qui sont rajoutées entre parenthèses.

5. Paquets et structures

Si les fichiers NED sont dans le répertoire `<root>/a/b/c` alors le nom du paquet est `a.b.c`. Par convention le nom des paquets est tout en minuscules. Ce nom doit être explicitement déclaré en haut des fichiers NED : `package a.b.c ;`

Les fichiers NED nommés `package.ned` jouent un rôle particulier. En effet, ils sont destinés à représenter l'ensemble du paquet. Il définit les sous-répertoires, les chemins. Les commentaires compris dans ces fichiers sont interprétés comme de la documentation. Il faut penser à rajouter (à déclarer) le fichier `package.ned` dans les fichiers qui en ont besoin.

Pour plus d'informations et des exemples cf. la partie 3.14 du document officiel.

6. Signaux

Pour plus d'information sur cette partie cf. partie 4.14.2 du document officiel.

Les signaux de simulation sont utilisés :

- Pour exposer les propriétés statistiques du modèle
- Recevoir des notifications sur les modifications du modèle de simulation pendant l'exécution
- Mettre en œuvre une communication de type publication-abonnement.

Les signaux sont émis par les modules ou les canaux. Ils se propagent sur la hiérarchie du module jusqu''à la racine.
Les méthodes liées au signal sont déclarées sur `cComponent` et sont disponibles pour `cModule` et `cChannel`. Les signaux sont déclarés avec la propriété `@signal` sur le module ou le canal qui l'émet.