PARAMETRIZED TOPOLOGIES FOR SIMULATION PROGRAMS

András Varga
Department of Telecommunications
Technical University of Budapest
H-1111 Budapest, Sztoczek u. 2.
Hungary
E-mail: vargaa@hit.bme.hu

KEYWORDS

Topology, Description Language, Hierarchical, Network, Computer Networks, Multiprocessors

ABSTRACT

The article proposes a language for the description of model topologies in discrete event simulators. The language contains an efficient way to create parametrized, flexible topologies. The language has been implemented as part of the OMNeT++ simulator.

In most simulators, the support for defining the topology of the model can be improved upon. For this task, simulators either (1) do not provide explicit support, or (2) only fixed topologies are supported, or (3) flexible topologies require programming.

The solution proposed and implemented in OMNeT++ uses a description language with a powerful combination of simple constructs (multiple connections, conditional connections etc.) to allow parametrized description of regular structures.

The paper presents several examples, introduces topology templates as a tool for reusing interconnection structure, presents three general patterns of using the tools of the language, and describes the issues of creating complex, parametrized structures in a simulation program at run-time.

INTRODUCTION

Discrete event simulators vary in the degree they support describing the topology of the model. The word topology will be used thoughout this paper to mean the fixed elements (nodes, resources etc.) and their interconnection structure.

Many simulators (e.g. Simscript (CACI 1983)) do not provide explicit support for this task; that is, topology is implicitly defined by program logic. This allows one to implement models with arbitrarily complex topologies, but at a cost of high program complexity and loss of clarity.

Other simulators support topology definition by providing a topology description language or/and a graphical editor. For example, the CLASS ATM simulator (Cigno and Munafò 1995; Marsan et al. 1995) has a description language, and most commercial simulators (e.g. OPNET (MIL3 1996), BONeS (Shanmugan 1992), etc.) feature graphical editors.

The common drawback of most of these tools is that they only allow creation of fixed topologies, with a fixed number of elements and a fixed interconnection structure. This limitation becomes painful when (a) the model contains a large number of elements with a regular interconnection structure or, (b) several simulation runs are needed with model topologies that are similar but differ in the number of elements or in the exact interconnection structure. In case (a), one is forced to prepare an unnecessarily large topology description that is difficult to maintain. In case (b), one needs to prepare a number of separate descriptions that are also difficult to keep in hand.

The above problem is usually solved by programming, where the code either (1) generates the appropriate topology description(s) (e.g. OPNET's EMA library), or (2) it becomes part of the simulation program and builds the model directly, at the beginning of the simulation. Note that here one completely loses the advantage offered by the description language or the graphical editor, again resulting in high program complexity and loss of clarity.

An adequate solution can be a description language which is flexible enough to handle the above problems, that is, which enables one to control the number and type of elements and their interconnection structure by means of parameters (Fig.1) (Varga and Pongor 1997).

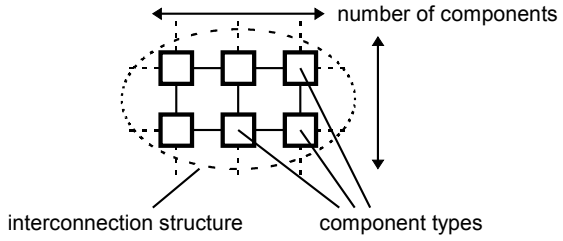

Fig.1. Aspects of the topology that should be allowed to be specified by parameters

The topology description language described in this paper intends to be such a language. The language has been implemented as part of the OMNeT++ simulator; a more detailed description about the language can be found in the OMNeT++ manual (Varga 1997). The manual and the complete source code of the simulator can be downloaded from the address given in the reference. For size limitations, however, the article does not discuss how the language was implemented.

MODEL STRUCTURE IN OMNET++

An OMNeT++ model consists of hierarchically nested modules which communicate with messages. OMNeT++ models are often

referred to as *networks*. The top level module is the *system module*. The system module contains *submodules*, which can also contain further submodules (Fig.2). The depth of module nesting is not limited; this allows the user to reflect the logical structure of the actual system in the model structure.

Modules that contain submodules are termed *compound modules*, as opposed *simple modules* which are at the lowest level of the module hierarchy. Simple modules contain the algorithms in the model and they are implemented by the user.

Both simple and compound modules in a given network are instances of *module types*. While describing the model, the user defines module types and uses them to define more complex module types. Finally, the user creates the system module as an instance of a previously defined module type; all modules of the network are instantiated as submodules and sub-submodules of the system module.

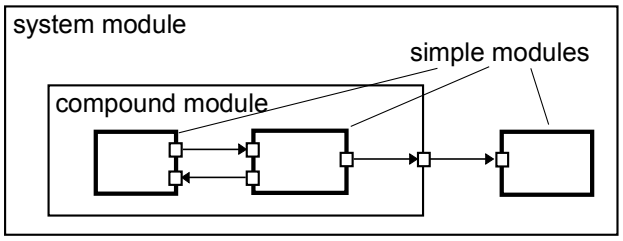

Fig.2. Model structure in OMNeT++: compound and simple modules, gates, connections

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

In an OMNeT++ model, modules communicate by exchanging *messages*. In an actual simulation, messages can represent frames or packets in a computer network, jobs or customers in a queueing network or other types of mobile entities. Messages are sent out and arrive through *gates*, which are the input and output interfaces of a module.

Input and output gates of different modules can be interconnected. Each *connection* is created within a single level of the module hierarchy: within a compound module, one can connect the corresponding gates of two submodules, or a gate of one submodule and a gate of the compound module (Fig.3).

Due to the hierarchical structure of the model, messages typically travel through a series of connections, to start and arrive in simple modules. Compound modules act as 'cardboard boxes' in the model, transparently relaying messages between their inside and the outside world.

Modules can have parameters. Parameters are used for two main purposes: (1) to customize simple module behaviour, and (2) to parametrize model topology. Compound modules can pass parameters or expressions of parameters to their submodules.

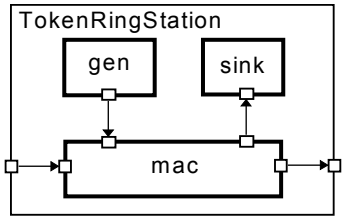

Fig.3. Submodules can be connected to each other or to the parent module

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 internal connections are made.

ELEMENTS OF THE TOPOLOGY DESCRIPTION LANGUAGE

An OMNeT++ model is defined by a textual network description. A network description contains declarations of simple module types, describes compound module types and contains a network definition that instantiates a compound module.

A simple module is defined by listing its parameters and gates:

```
simple TokenRingMAC
  parameters:
    THT, address;
  gates:
    in:  from_higher_layer,
         from_network;
    out: to_higher_layer, to_network;
endsimple
```

Compound modules are modules composed of one or more submodules. Submodules can be either simple or compound modules. A compound module description - in addition to parameters and gates - also specifies the submodules and the connections within the module. For example, the compound module in Fig.3 can be defined by following code:

```
module TokenRingStation
  parameters:
    mac_address;
  gates:
    in: in; out: out;
  submodules:
    mac: TokenRingMAC
      parameters: THT=0.010,
                  address=mac_address;
    gen: Generator;
    sink: Sink;
  connections:
    mac.to_network --> out,
    mac.from_network <-- in,
    mac.to_higher_layer --> sink.in,
    mac.from_higher_layer <-- gen.out;
endmodule
```

An actual simulation model (a system module) is created as an instance of a module type:

```
module TokenRing //...

network token_ring: TokenRing
  parameters: num_of_stations = 12,
              load = input;
endnetwork
```

In the OMNeT++ simulator, the textual model description is compiled into C++ code and linked into the simulator executable. This makes it very fast to build up the model internally when the simulation starts.

ELEMENTS SUPPORTING FLEXIBILITY

Module And Gate Vectors

As the first step towards flexibility, the language allows one to create *a vector of submodules* within a compound module. The notation is to specify the vector size -- which can be a constant, a parameter or an expression of parameters -- in brackets (e.g. `TokenRingStation[10]`). The elements of the vector are typically, though not necessarily, identical.

It is also possible for a simple or a compound module to have vectors of gates. The size of the gate vector -- also a constant or an expression -- can be specified inside the module description or outside of it. In the second case, the module description contains only an empty bracket pair to declare the gate vector (e.g. `inputs[]`), and the actual vector size is specified by a `gatesizes` section in a compound module description in which the module is used as a building block.

Multiple Connections

If submodule vectors and/or gate vectors are used, the language allows one to create more than one connections with one statement. This is termed a *multiple* or *loop connection*. Multiple connections are created with the `for` statement in the `connections` section of a compound module description. One can place several connections in the body of the `for` statement, separated with semicolons. `for` statements can also be nested; this is done by specifying more than one indices in a `for` statement, with their own lower and upper bounds. Typical basic uses of multiple connections is to create one-module-to-many connections and to connect modules into a chain.

Submodule vectors, gate vectors and multiple connections are illustrated in the following example:

```
simple Hub
  gates:
    out: outport[];
endsimple

simple Station //...

module Star
  submodules:
    hub: Hub
      gatesizes: outport[4];
    station: Station[4];
  connections:
    for i=0..3 do
      hub.outport[i] --> station[i].in;
    endfor
endmodule
```

The result of the above is depicted in Fig.4.

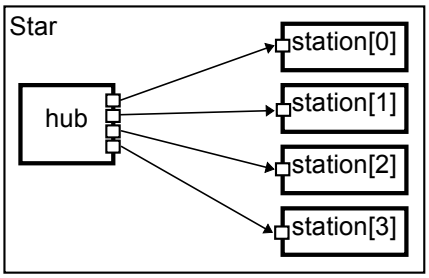

Fig.4. Multiple connections created using `for`

Conditional Connections

Connections can be made conditional by tagging them with an `if` clause. Conditions can use module parameters and indices from the enclosing `for` statement.

Conditional connections combined with `for` enable one to create more sophisticated topologies. An important usage is to generate random topologies. The following code generates a random subgraph of a full graph:

```
module RandomGraph
  parameters:
    count,
    alpha; //connectedness: 0.0<x<1.0
  submodules:
    node: Node [count]
     gatesizes:
       in[count], out[count];
  connections nocheck:
    for i=0..count-1, j=0..count-1 do
      node[i].out[j] --> node[j].in[i]
        if i!=j and uniform(0,1)<alpha;
    endfor
endmodule
```

Note that in the `RandomGraph` example, not every gate of the modules will be connected. By default, an unconnected gate produces a run-time error message in OMNeT++ when the simulation is started, but this error message is turned off here with the `nocheck` modifier. Consequently, it is the simple modules' responsibility not to send on a gate which is not leading anywhere.

Conditional Sections

It is often needed that elements of a submodule vector have different parameter values and gate vector sizes. One way to achieve this is to include the submodule index (`index` keyword) in the expressions of the parameter values or gate sizes. Another way is to use multiple `parameters` and `gatesizes` sections within a submodule definition, each section tagged with conditions.

The following example shows the use of conditional `gatesizes` sections in creating a chain of modules. Different

gate vector sizes are defined for the modules at the ends of the chain which have only one neighbor.

```
module Chain
  parameters: count;
  submodules:
    node : Node [count]
      gatesizes if index==0 or
                   index==count-1:
        in[1], out[1];
      gatesizes:
        in[2], out[2];
  connections:
    for i = 0..count-2 do
      node[i].out[i!=0 ? 1 : 0] -->
                        node[i+1].in[0];
      node[i].in[i!=0 ? 1 : 0] <--
                        node[i+1].out[0];
    endfor
endmodule
```

A similar solution could be used for defining a mesh, where different number of gates are required for modules inside the mesh and along the edges.

Submodule Type As Parameter

Instead of supplying a concrete module type for a submodule, one can leave it as a parameter. At the same time, to let the compiler know what parameters and gates that module has, the user has to supply the name of an existing module type. This is done with the `like` phrase.

```
simple GeneralNode //...

module CompoundModule
  parameters:
    node_type;
  gates: //...
  submodules:
    theNode: node_type like GeneralNode
      parameters: //...
  connections: //...
endmodule
```

The above example means that the type of the submodule `theNode` is not known in advance; it will be taken from the `node_type` parameter of `CompoundModule` which must be a string (for example, "`SwitchingNode`"). The module type called `GeneralNode` must have been declared earlier; the declaration will be used to check whether `theNode`'s parameters and gates exist and are used correctly. The `node_type` parameter can be given a value at a higher level in the module hierarchy or can be specified in an external configuration file. The `GeneralNode` module type does not need to be really implemented because no instance of it is created; it is merely used to check the correctness of the network description.

As an example of usage, if one wants to drive a model of a computer network with different traffic generators, he can leave the type of the generator modules as parameter and specify it individually for each run.

The `like` phrase encourages the user to create *families* of modules that serve similar purposes and implement the same interface (they have the same gates and parameters) and can be used interchangeably. This scheme directly parallels with the concept of *polimorphism* used in object-oriented programming. The `like` phrase also plays a cental role in *topology templates* (to be discussed in the next section).

TOPOLOGY TEMPLATES

Leaving submodule type as parameter enables the user to create *topology templates*: compound modules that implement mesh, hypercube, butterfly, perfect shuffle and other topologies (Moldovan 1993), with the actual element type left as parameter. Topology templates are a means of reusing interconnection structure.

The concept is demonstrated on a network with hypercube interconnection. When building an *N*-dimension hypercube, we can exploit the fact that each node is connected to *N* others which differ from it only in one bit of the binary representations of the node indices (Fig.5).

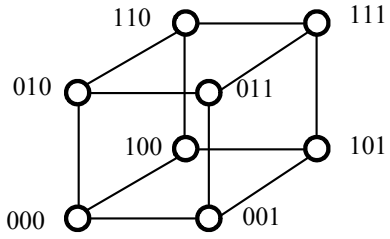

Fig.5. 3-D hypercube interconnection

The hypercube topology template is the following (it can be placed into a separate file, e.g `hypercube.ned`):

```
simple Node
  gates: out: out[]; in: in[];
endsimple

module Hypercube
  parameters:
    dim, nodetype;
  submodules:
    node: nodetype[2^dim] like Node
      gatesizes:
        out[dim], in[dim];
  connections:
    for i=0..2^dim-1, j=0..dim-1 do
      node[i].out[j] -->
                  node[i binxor 2^j].in[j];
    endfor
endmodule
```

When the user creates an actual hypercube, he substitutes the name of an existing module type (e.g. `Hypercube_PE`) for the `nodetype` parameter. The module type implements the algorithm the user wants to simulate and it must have the same gates that the `Node` type has. The topology template code can be used through file inclusion.

```
include "hypercube.ned"

simple Hypercube_PE
  gates: out: out[]; in: in[];
endsimple

network hypercube: Hypercube
  parameters:
    dim = 4,
    nodetype = "Hypercube_PE";
endnetwork
```

Another popular interconnection structure is the binary tree. The following topology template creates a binary tree of arbitrary height from nodes shown in Fig.6.

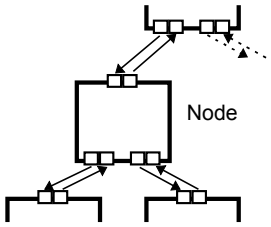

Fig.6. Binary tree node

Nodes of the binary tree are mapped to module vector indices like in the heap data structure: *node[0]* is the root element and *node[i]*'s children are *node[2i+1]* and *node[2i+2]*.

```
module BinaryTree
  // node gate indices: 0=up 1=left 2=right
  parameters:
    height, nodetype;
  submodules:
    node: nodetype[2^height-1] like Node;
      gatesizes:
        out[3], in[3];
  connections nocheck:
    for i=0..2^(height-1)-1 do
      node[i].out[1] --> node[2*i+1].in[0];
      node[i].in[1] <-- node[2*i+1].out[0];
      node[i].out[2] --> node[2*i+2].in[0];
      node[i].in[2] <-- node[2*i+2].out[0];
    endfor
endmodule
```

CONNECTION PATTERNS AND GENERALITY

Several approaches have been identified that can be applied when creating complex topologies with multiple conditional connections.

'Subgraph of a Full Graph'

The following pattern creates a subset of connections of a full graph. A condition is used to "carve out" the necessary interconnection from the full graph.

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

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

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

'Connections of Each Node'

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

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

From the examples presented earlier, the `Hypercube` compound module is a clear example of this approach. `BinaryTree` can also be regarded as an example of this pattern where the inner *j* loop is unrolled.

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

'Enumerate All Connections'

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

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

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

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

In the case of irregular structures where none of the above patterns can be employed, the user can resort to specifying constant submodule/gate vector sizes and explicitly listing all connections, like he/she would do it in most existing simulators.

DYNAMIC MODULE CREATION AND TOPOLOGY OPTIMIZATION

In the OMNeT++ simulator, one can create modules at run-time and connect their gates to existing modules. As mentioned earlier, in OMNeT++ the textual model description is compiled into C++ code and linked into the simulator executable. This fact has the following important consequence: when one creates a compound module dynamically, its submodules and internal

connections can be automatically built out by the compiled network description code.

The solution implemented in OMNeT++ also allows one to set the parameters of a to-be-created compound module. For example, if the description of the `BinaryTree` module type (see an earlier section) has been compiled and linked into the simulation executable, one will be able to create any number of `BinaryTree`s at run-time, with run-time-selected tree heights and node types. The way this feature can be used is outlined in the following pseudocode:

find description object for "BinaryTree";

ask the description object to create an empty compound module;

set the new module's "height" parameter to 5;

set its "nodetype" parameter to "BinTree_PE";

ask the description object to build out the internals (submodules, connections) of the new module.

The last step calls the C++ code that `BinaryTree`'s description was compiled into.

The solution opens up the possibility for a wide range of simulation experiments: those where the topology changes at run-time, or even, models are created at run-time. One group of questions that can be more easily answered using this feature sound like: "How many stations with the given characteristics can be attached to this LAN?" or "How do the traffic conditions change in this WAN if I add subnetworks of different sizes and characteristics?"

The feature is useful also when model topology itself is to be optimized by simulation. An example problem: "What is the best interconnection structure of a fine-grain multiprocessor to optimally execute this parallel algorithm?"

SUMMARY

The paper introduces the model topology description language of OMNeT++. The language allows one to create parametrized descriptions of regular structures. The model description is modular: the user builds new module types out of simpler ones. The key idea of the description language is that properties of the topology (e.g. number of nodes, interconnection structure) should be able to be controlled by parameters. To facilitate this idea, submodule vectors and gate vectors were introduced. Parameters can be used to specify submodule types, sizes of submodule and gate vectors and also to describe multiple (or loop) connections.

It was demonstrated that using the tools provided by the language, one can create regular structures such as star, chain, binary tree, hypercube etc. in an easy way. The concept of topology templates was introduced which are a tool for reusing interconnection structure. Three general patterns were presented that make it possible to generate any topology that can be described by mathematical formulas, including random topologies.

The implementation of the language allows one to easily create parametrized structures at run-time. This means that one can perform experiments where the model topology changes during simulation; one can even optimize the model topology within a single simulation run.

The description language uses simple constructs which make it easy to implement. In the OMNeT++ simulator, run-time efficiency is guaranteed by the fact that the textual model description is translated into C++ and used in its compiled form. This makes it very fast to build up the model internally when the simulation starts.

REFERENCES

CACI Products Company. 1983. *SIMSCRIPT II.5* Programming *Language*. La Jolla, CA.

Cigno, R.L.; M.Munafò. 1995. "RC - A Flexible Language for the Specification of ATM Networks Simulation Experiments." In *Proceedings of the Third Workshop on Performance Modelling and Evaluation of ATM Networks.* (Ilkey, UK, July 2-6).

Marsan, M.A.; A.Bianco; T.V.Do; L.Jereb; R.L.Cigno; and M.Munafò. 1995. "ATM Simulation with CLASS." *Performance Evaluation* 24: 137-159.

MIL 3, Inc. 1996. *OPNET Modeling Manual, Release 3.0*, Washington, DC.

Moldovan, D.I. 1993. *Parallel Processing, from Applications to Systems*, Morgan Kaufmann, 191-204.

Shanmugan, K.S.; V.S.Frost; W.LaRue. 1992. "A Block-Oriented Network Simulator (BONeS)." *Simulation.* (Feb.).

Varga, A. 1997. *OMNeT++ User Manual*, Dept. of Telecommunications, Technical University of Budapest. http://www.hit.bme.hu/phd/vargaa/omnetpp.htm

Varga, A.; Gy.Pongor. 1997. "Flexible Topology Description Language for Simulation Programs". In *Proceedings of the 9th European Simulation Symposium*. (Passau, Germany, October 19-22).