

Semla tutorial

Rikard Thid

8th December 2003

1 Introduction

Semla is a network on chip simulator based on SystemC. It is assumed that the reader of this document is familiar with SystemC.

2 Semla

2.1 Overview

The Semla is a layered network simulator. Different functions are modeled in different layers, the ones implemented are:

- Transport Layer (TL)
- Network Layer (NL)
- DataLink Layer (LL)
- Physical Layer (PL)

The Transport layer handles messages, which are transferred between different *Resources*. Messages are segmented into packets which is the unit that is transferred at the Network layer. The Network layer handles routing and addressing of packets, transferred between switches/routers at the Datalink layer. The physical layer is where physical aspects such as error introduction is handled.

At each layer, the functionality is handled by the interaction of *Entities*. Basic entities for the Nostrum architecture are included in the source directory. The base class for all models is the `ent_base`. It defines the very basics of how an entity looks like. For instance it is defined that an entity has a number of *inports* and *outports*. Also there are some help methods, such as `is_inport_empty()` which tells whether or not data is available on the inport. All entities are indirectly inherited from a few base-classes, specific to what layer they implement as depicted in Figure 1.

A semla model of a network is possible to connect to a systemc model of an application through an *interface*, which specifies which communication functions (or *primitives*) the network provides (e.g. read and write).

copying of data, and therefore the speed increases. Furthermore the PDUs are flexible, you can remove and add data elements to a PDU in another part of the model than that it was created in.

Figure 3 illustrates a part of the functionality in the Nostrum stack. We will use this example to look at how data entries are read and written from the PDUs. This specific example shows how address translation is handled in the Nostrum stack, but the examples apply to all access to PDUs.

In the Nostrum stack, the communication is handled at a transport layer in different *channels*. The transport layer uses the network layer in order to realise these channels, but in nostrum, we do not want to deal with *channel ids* at the network layer, for efficiency reasons. Therefore we must translate the channel ids to a *destination address*. The destination address has an x and an y component, which the stack puts in different data entries.

In the `nl_example` implementation of the network layer, information is read as follows:

```
u->read_int(CID);
```

Where `u` is a pdu, `read_int` is a method of this pdu. `_int` specifies the datatype of the pdu and `CID` is defined to `6`¹ an integer that indicates which data entry is to be inserted. Now, the will layer will translate the cid to destination address, how this is done exactly is out of the scope of this tutorial. However, the destination address is added to the pdu in the following way.

```
u->insert_int(NL_DEST_ADDR_X, x);
```

```
u->insert_int(NL_DEST_ADDR_Y, y);
```

Where `u` is the very same pdu as used above, `NL_DEST_ADDR_X` and `NL_DEST_ADDR_Y` are integers that identify the data record. `x` and `y` are the integer values that we wish to set these records to. Again, the result of the performed methods are depicted in Figure 3.

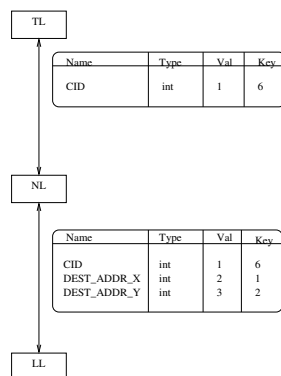


Figure 3: PDU

2.4 Interface

SystemC supports *channel refinement* which is a way to isolate the various parts of a system from its communication infrastructure. In the Nostrum model, the resources is

¹This is defined in the file `tl_example.h`

what makes up a system and the network is the communication infrastructure. Figure 4 depicts the way in which a system is composed on a system level. Each resource model must have a *Port*, which allows it to access the network. This port is connected to an interface of a NoC model. The interface is defined in the file `mp_if.h`. The communication is message passing based and features both blocking and non-blocking types of read and write primitives. All read and write primitives are performed on channels which are opened with the `open_channel` primitive.

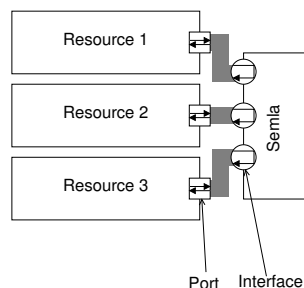


Figure 4: Resources are connected to a NoC model via its interfaces.

3 The Semla environment

3.1 Library structure

3.2 Setting up the environment

3.3 Obtaining Semla

Since Semla is located in a CVS repository you must configure your environment before you can check out versions of semla. The reader should possess some rudimentary knowledge of CVS before continuing.

2

- To specify which repository to use, the environmental variable `CVSROOT` must be set. This is done by the command lines marked with `csh`: below in your `csh/tcsh` shell (or the lines that follow `sh`: if you use `sh`, `ksh` or `bash`). It is recommended that you add the lines that sets environment variables in your shell setup file (which may be named `.bashrc`, `.cshrc`, `.tcshrc` or something else depending on what shell you use). Find out which username that you have on the repository, if you don't have one you should mail the author of this document about it:


```
csh: setenv CVSROOT :pserver:your_user_name@cvs.imit.kth.se:/localdisk/cvs-
rep/
semLa
```

```
sh: CVSROOT=:pserver:your_user_name@cvs.imit.kth.se:/localdisk/cvs-
rep/semLa export CVS_RSH CVSROOT
sh:export CVSROOT
```

²We recommend Open Source Development with CVS. <http://cvsbook.red-bean.com/>

- To specify what editor to write repository messages with, type:

```
csh:setenv CVSEEDITOR emacs
```

```
sh: CVSEEDITOR = emacs
```

```
sh: export CVSEEDITOR
```
- Most importantly, you must always use ssh to connect to the server. Therefore write:

```
csh: setenv CVS_RSH ssh
```

```
sh: CVS_RSH = ssh
```

```
sh: export CVS_RSH
```
- Log in to the repository:

```
csh/sh: cvs login
```

You will be prompted for password which could be retrieved from the author of this document.
- Make a sandbox directory, it is recommended that you create this in your home directory:

```
sh/csh: mkdir ~/sandbox
```
- Set the sandbox as your current directory:

```
sh/csh: cd sandbox
```
- Check out the latest version of Semla by writing:

```
sh/csh: cvs checkout Semla
```

To list the files, type:

```
sh/csh: ls Semla -p
```

The Semla directory should look something like this:

```
CVS/ experiments/ semla.tar.gz
doc/ Makefile.defs source/
```

`Makefile.defs` contains pointers to the SystemC environment. The semla core is located in the `source` directory. Here, extensions to semla should be placed. When experiments are made by a user, these should be put in the `experiments` directory. Examples of such experiments include modification of the workload, routing algorithms, error models, etc.

3.4 Compiling Semla models

The Semla models must be compiled with a c++ compiler before run. Make sure that SystemC is installed and referred to correctly in `Makefile.defs`. The compiler that is preferred is gcc-2.95.3³.

Now, lets try to compile the semla classes.

³On the imit computers, this version is invoked by writing: `tool gcc`.

- Log in to a powerful computer, compiling SystemC-models typically requires >200 MB of free memory:
`>telnet servername`
- Go to the Semla/source directory in your sandbox:
`>cd ~/sandbox/Semla/source`
- The simulator is built by typing:
`>make`
 this should take a few minutes and generate some warnings but hopefully no errors.
- Now you should have a file called run.x.
`>run.x`
 The simulation takes a few seconds and ends by printing statistics of how the NoC model performed. It is up to the designer to define, interpret and collect the statistics in a simulation. (More to come in later chapters....)

4 Working with Semla - A Set of Tutorials

Lets take a look at how we modify the existing semla model to report statistics on for instance, the activity level in the physical layer. A dedicated directory for your own modified version of the semla environment is necessary if you are not familiar with cvs.

Let *MYDIR* be the directory in which you make your changes to the simulator, and let *SANDBOX* be the directory in which you have your cvs sandbox. Make a copy of the semla simulator to your own directory:

```
~>mkdir MYDIR
~>mkdir MYDIR/source
~>cp -r SANDBOX/Semla/source MYDIR
~>cp SANDBOX/Semla/Makefile.defs MYDIR
```

4.1 Modifying the Network

Now, lets modify the source code of the `meshtop.h`, which is one of the files where the topology of the network is defined. In here there is a line that looks something like:

```
typedef pl_void pl_type;
```

This line defines the type `pl_type` to be replaced by `pl_void` everywhere it occurs. `pl_type` also occurs in `meshtop.cpp`, which contains the actual routines that instantiates the mesh topology (and the entities that it is made up form).

Let's take a closer look at `pl_void.h` which is a header file. In Semla all modules have their declarations in the headers⁴. Notice that the constructor does not initialize any variables and the destructor is empty, due to the simple nature of the model. When a `pl_void` is constructed, it must have only one inport and one outport. This is defined when the constructor is called with the `int n_in_ports` and `int n_out_ports` arguments. The one process that is declared here is `void pl_void::process()` which is sensitive to the `in[0]` port, meaning that `process()` is called whenever `in[0]` is changed. Now, let's take a closer look into that particular process and see what it does:

⁴this is not necessarily the case in other SystemC models

```

void pl_void::process() {
    out[0]->write(in[0].read());
}

```

This very line just writes what is on the inport to the output.

Suppose we want to do some measurement of how much every single link⁵ is used. Then we must change model of the physical layer. Again, in `meshtop.h` you find the line `typedef pl_void pl_type;`

Change it to: `typedef pl_count pl_type;`

The `pl_count` model has two variables that are increased every time new data is transmitted over a link. `local_active` is a variable that is specific to every different link and is increased every time data is transmitted over it. `global_active` is a static variable, which means that it is shared among all different links in the simulation. It is also increased whenever new data arrives over any link, and since it is shared, it is the sum of activity all over the links.

In the file `pl_count.h` the constructor sets these variables to zero at the beginning of the simulation:

```

global_active=0;
local_active=0;

```

In the destructor the data that was collected during simulation is displayed:

```

cout << name << ": " << local_active << endl;
if(global_active!=0) { /* Make sure the global variable is only written
                        once. */
    cout << " global_active pl: " << global_active << endl;
    cout << "MATLAB style:" << endl;
    cout << "gl_active_pl = [gl_active_pl, " << global_active << "];"
        << endl;
    global_active=0;
}

```

In the `pl_count.cpp` file, the behavior is defined:

```

void pl_count::process() {
    pdu *x;
    x=in[0].read();
    if(x!=NULL) {
        global_active++;
        local_active++;
    }
    out[0]->write(in[0].read());
}

```

Here, the pdu on the inport is stored to `x` and if the inport actually contained something, the counters will be increased.

⁵A link is a bunch of wires, the entity of the physical layer.

4.2 Retrieving bit level data from simulation

It is also possible to get the actual data logged to a file by changing the `pl_type` to `pl_log`, which is defined in `pl_log.h` and `pl_log.cpp`.

4.3 Modifying the Application