Dynamic networks

NetGen : objectives, installation, use, and programming

Bernard Pottier
Université de Bretagne Occidentale

September 4, 2013

Chapter 1

Installation and first experiments

1.1 Smalltalk, the underlying development system

NetGen has been developed using Smalltalk, a powerful object-oriented language. Smalltalk is available on different platforms such as: VisualWorks, from Cincom, VisualAge, from IBM, Pharo, coming from free software community.

Historically, the language has been created at Xerox PARC, and divulged with precise specifications of its syntax, intermediate code and tools. This has allowed universities and growing IT companies to implement virtual machines, or even hardware to execute the virtual image published by the PARC.

Visualworks is a branch coming from the initial version with a lot of improvements and the capability to follow software and system progresses due to fast integration tools.

Pharo, the 'free Smalltalk' is completely a new design.

Most of the Smalltalk environments are interpreted, and thus, executed by a *Virtual Machine* (VM). The VM is processor and system dependent. The object environment is located in large files called *Virtual Images*, because they reflect the abstraction of the object organization. Images are deployed inside the computer memory at run-time, and they are dumped into files to be restored when useful. At the difference of a VM, the virtual image is more or less platform independent.

Practically, to run a Smalltalk environment, a user need to apply the VM to an image.

An application written in Smalltalk is simply a dedicated image prepared by developers that is executed by a VM. This dedicated image does not have development tools and appears exactly as a normal application to an end user.

NetGen, the software presented in this report, can be seen as an application. Due to the commercial nature of Visualworks, the only choice was to distribute as package, and let interested users to load them on a standard image.

1.1.1 What is needed

To work with NetGen it is necessary to prepare a specific environment:

1. **VisualWorks VM** : as distributed by Cincom
2. **Visualworks image** : also from Cincom. These two items are installed from *personal use*, *non-commercial distributions*, available on http://cincomsmalltalk.com.
3. **NetGen packages** : downloaded from a server at University of Brest. A running VisualWorks system is necessary to access the data base on http://wsn.univ-brest.fr.

As a benefit from VM technique, it is possible to run the software on common platforms: Linux, MacOSX, Windows. However, external software/compilers are used as a target. Integration of these tools in the design flow necessitates:

1. **kroc** : the Occam compiler from university of Kent. Kroc provides a concurrent process oriented environment that can execute network simulation on multi-core processors. Basically, networks are transformed on communicating process syntax, one thread per node.

2. **CUDA** : the Nvidia environment for Graphic Processing Units (GPUs) on which networks are mapped, one thread per node, communications executed in shared memory.

3. **Graphviz** : a well known network graphical presentation package that allows to draw networks for documentation, as example.

1.2 VisualWorks installation

The Cincom site proposes an evaluation ISO file for download, with a non commercial license. Read the statements and download the CDrom (if you agree).

After this, it comes a 600Mo .iso file that can be used on your platform (we prefer Linux). This file must be mounted as a fake CDrom, generally by simply clicking the file icon. Installation is done by following the default choices of the CDrom. It can be a good idea to setup the files in a system place rather than ones home directory (as example, /Developer, on MacOSX.

On Linux, it is possible to proceed in from a terminal command line by saying:

```
sudo mount -o loop,exec  CSTxx.iso /mnt
```

CSTxx.iso is replaced by the name of the downloaded file on the system, and /mnt is the local directory where the CDrom will appear (`ls /mnt` will show the installation files). After this, you will type `/mnt/installUnix` and follow the instructions.

32 bits or 64 bits? . As the processors are evolving, it was also necessary to evolve VMs to follow these progresses. On Linux, it is necessary to be aware of the system characteristics (type `uname -a`). **kroc** is still 32 bits, thus the best choice would be to remain with a 32 bits virtual machine, and virtual image.

1.3 Creating an initial environment

The last versions of VisualWorks propose project folder as a convenient way to manage different development, thus different image. On MacOSX, a folder appears on the desktop that provides direct access to different environments.

On Linux, our practice is to proceed in the following way:

1. **install csh**: `apt-get install csh`, csh is used to interface Linux or MacOS at the command level,

2. **create a project directory**: `mkdir project1 ; cd project1`

3. **locate VisualWorks**: directory where you put VW during its installation. As examples, `/usr/local/vw7.9.1pu` for a system installation, or `/home/myname/vw7.9.1pu` for an installation at `myname` home directory.

4. **create a script** command to start a new image. Call it `startInit` to recall that it start an initial environment. The script is for *bash*, to setup an environment variable, then to launch the virtual machine executable *visual*, on the initial virtual image *visualnc.im*.

Listing 1.1: bash version

```
#!/bin/bash
export VISUALWORKS=/usr/local/vw7.8.1nc
echo $VISUALWORKS
${VISUALWORKS}/bin/linux86/visual ${VISUALWORKS}/image/visualnc.im
```

This script is for the 7.8.1 Visualworks home installed at the system level, and not the `/usr/local/vw7.9.1pu` that could be the choice for a recent VisualWorks. The VISUALWORKS variable is setup to point to this home. It is used inside Smalltalk to access lot of resources. Don't miss to configure it correctly!

5. make the script executable, and run it:

   `chmod +x startInit ; ./startInit`

If everything is fine, the VM is up, showing two windows figure 1.1.

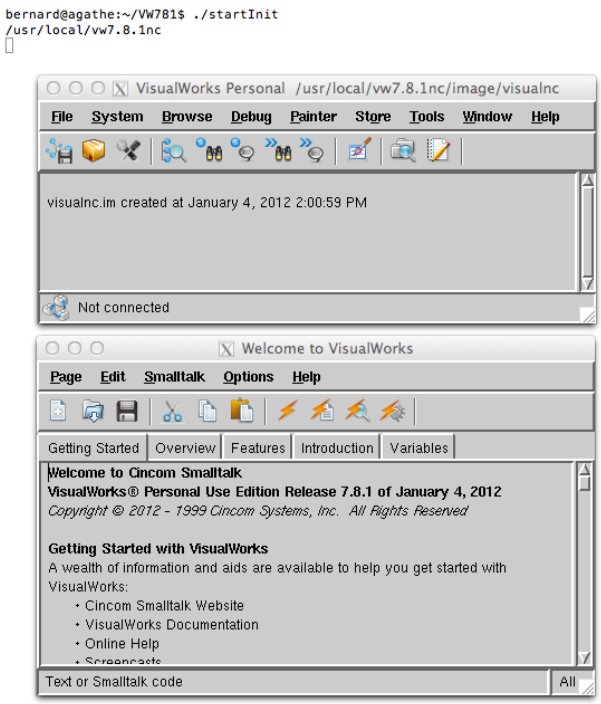

Figure 1.1: Initial environment

1.4 Creating a new project

1.4.1 New image file creation

The process of creating a new environment is reproduced for each new project. Once the initial environment is up, we save it to the new project name.

The trick is simply to save the image at the current location, or to a newly created directory, thus creating a new image file. Figure 1.2 shows an image creation Dialog opened from *File* > *save as* menu. Notice the following points:

- Access to the current directory by the right button on the first line of Dialog. The default location shown on the button is the VISUALWORKS home which is not suitable as a working location.

- the image file name is changed to `project1.im` to reflect the name of a new project.

After this, do a *Save*, then using *File* > *Exit*, quit the initial image without saving.

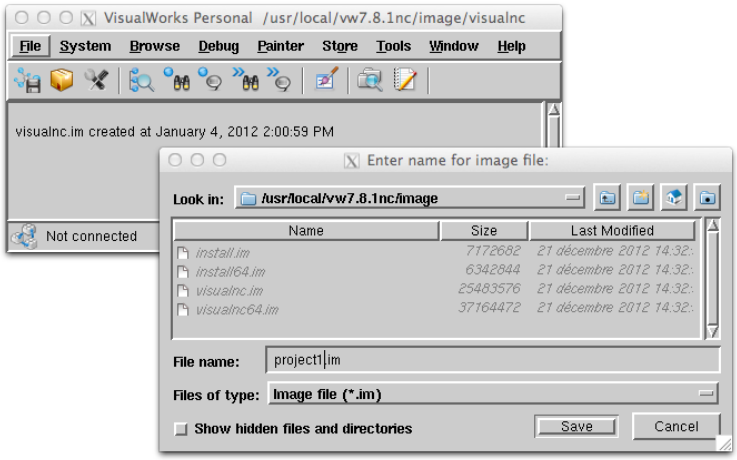

Figure 1.2: Creating a new image file *project1.im*

1.4.2 New script creation

Now, we have a new file called *project1.im* that we ca use to host developments. Check its presence by listing the directory `ls project1.*`.

It is more easy and secure to setup a new script file to launch our project. So, we copy startInit to startProject1, (`cp startInit startProject1`), and we modify startProject1 as follows:

Listing 1.2: bash version

```
#!/bin/bash
export VISUALWORKS=/usr/local/vw7.8.1nc
echo $VISUALWORKS
${VISUALWORKS}/bin/linux86/visual ./project1.im
```

Then, as in section 1.3, `chmod +x startProject1 ; ./startProject1`, that launch the new project safely.

1.4.3 Summary

- Creation of a directory to host developments
- Two scripts to launch the initial environment, and to launch a new project environment

We just need to recall the files location, we can launch and quit the project, making the choice of saving modifications to a file or not.

1.5 Connecting to Store

Once the *project1* environment is launched, it becomes possible to connect to software repository. This is done by the Store facilities in the main window. Observe the *Store* menu.

1.5.1 Accessing a repository

Figure 1.3 shows the Dialog allowing to connect to a package repository. The menu *Store* > *Connect Repository* will open it. The fields are filled with location and permissions to access the server at UBO. It is safe to save the connection to allow an easy reuse. Connection is normally fast, and release the Store menu quickly.

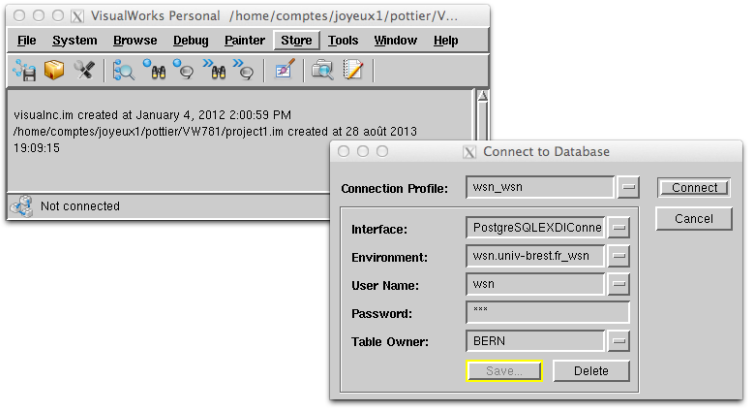

Figure 1.3: Defining a repository access

1.5.2 Loading packages

Once a connection is valid, by using *Store* > *Published items*, we can observe the database contents, select package, an load software. Figure shows how to proceed in the case of DistributedModeling:

1. select the name of a package on the left list
2. watch the different versions appearing in the right list
3. select one version
4. open the version menu and says *Load*

The Store tool will retrieve packages and needed dependencies from the server (if these dependencies are correctly defined). This take time. At the end of the process, the NetGen window appears (figure 1.5). Th Hotdraw window can be closed safely, this package is of marginal interest in the project.

1.5.3 Checking NetGen

By selecting 2D Random, and Graphic layout, then by pushing the Generate button, a random layout of 20 systems is produced, and connections are computed based on a minimum distance of 100 points. Figure 1.6 shows a different case, where the number of systems has been increased to 40. The graphic window displays the resulting layout, with 5 networks and 3 isolated nodes: the bottom left view inside the control window states that 37 nodes are connected.

The edges in the graph represents possible connections between nodes, given a range of 100. Grey zone figures uncovered points, while white zones are always under control of a node.

The full statistic for this sample quantifies the graph structure in relation with a surface where the network is produced.

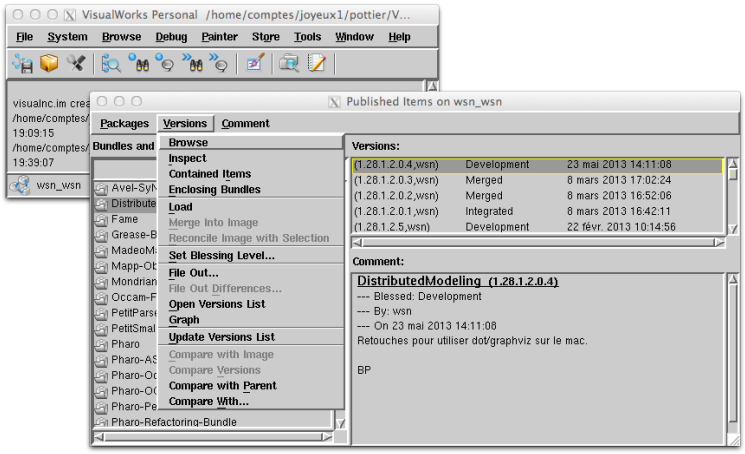

Figure 1.4: Choosing and loading packages

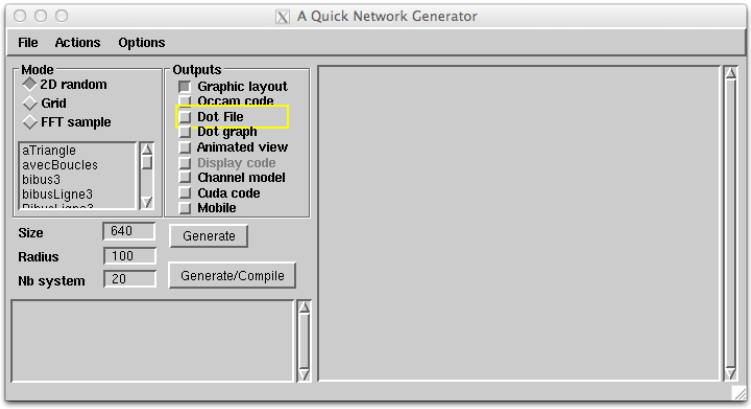

Figure 1.5: NetGen initial window

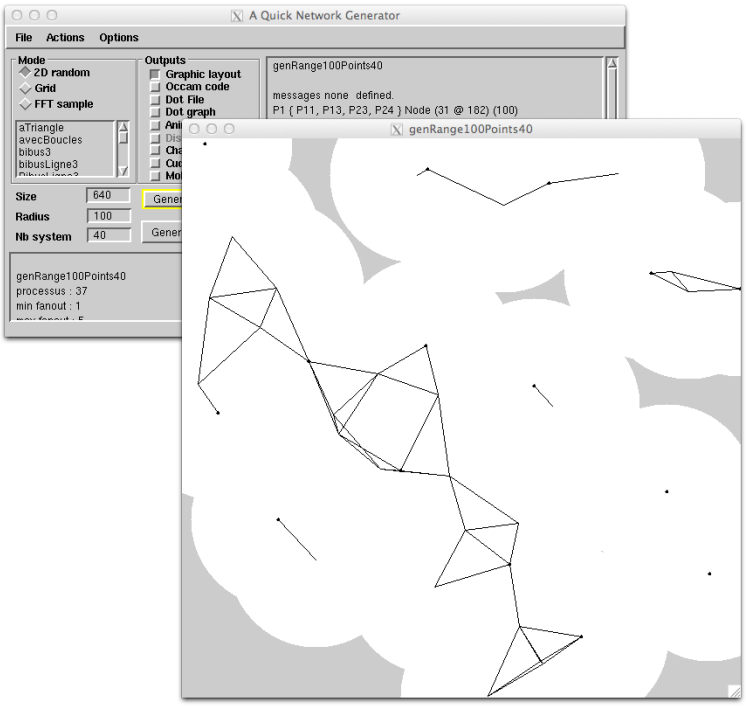

Figure 1.6: NetGen : random network generation with 40 nodes

```
genRange100Points40
processus : 37
min fanout : 1
max fanout : 5
channels   : 114
coverageArea   : 368718
percentArea   : 90
```

1.6 Summary

1.6.1 Knowledge status

At this point, it is probably useful to save an image from the VisualWorks launcher window : *File* > *Save*. Knowledge status is :

1. Development tools installation for VisualWorks,
2. Connection to a package repository data base
3. Loading NetGen development tools
4. Checking NetGen functionality

1.6.2 More background, some useful tricks about Smalltalk

- The selection of *Tools* > *Workspace* inside the main window launcher, launches an additional window similar to a terminal. Inside this window, users can type and execute Smalltalk statements.

Execution is obtained by selecting a piece of code, calling a pop-up menu (right click), selecting *doit*, or *printit* (or *inspect*) inside this menu. Pay attention to the fact that the menu must be observed carefully to do a selection. By releasing the *printit* option, the code is compiled, evaluated, and the resulting value in variable *x* is displayed in the Workspace.

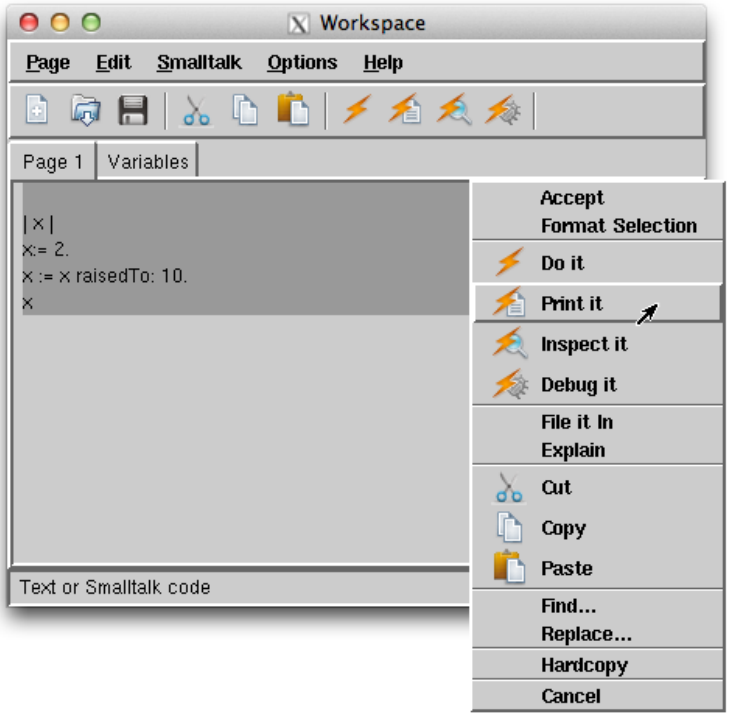

Figure 1.7: Programming in work-spaces

- In a similar way, application windows can be opened, or data set can be prepared interactively. To launch a window similar to figure 1.6 , type `UINetworkGeometry open` in a Workspace, and call *doit* from the pop-up menu.
- More about Smalltalk programming : the System Workspace window (figure 1.2) gives access to lot of contributions about the language and the system. The Smalltalk syntax is very simple, thus easy to learn: *get started...*

Chapter 2

Building abstract networks

Our networks are abstractions appearing basically as graphs, grouping *nodes* representing actual systems, *edges* representing communication links. Abstraction allows to cover lot of situations, from the nano scale to the universe, and lot of domains, such as distributed sensing, distributed computing, communication systems, environment modeling, bio systems.

Here, the focus is on wireless sensor networks design and validation in regards of practical situations in the environment. NetGen software *upper services* have the following objectives:

1. Practical system description based on geometry.

   As example, from a map, one can decide sensor locations taking into account physical considerations, decide on a wireless technology, and infer workable communication links.

   Description can be achieved based on maps, or pictures. Alternatively, generators allow to produce random distributions of different characteristics. A Text input format allows to exchange network topologies with external tools. .

2. Behaviour description.

   As example, nodes will execute programs, (1) to control and sense locally physical phenomena, (2) to contribute to the distributed system activities, such as collecting, transforming data, sending alerts.

   Behaviour description follows currently the synchronous model that use discrete time boundaries to make system evolutions. This technique is relevant for most of the sensor network technologies, as example 802.15.4, or commercial Digimesh. This is because sensor systems need to go to sleep and wake up periodically..

3. Validating a system behaviour.

   As example, the communication load implied by a particular topology, the latency and cost for producing diagnostics, energy cost of a particular algorithm, risks of failures, redundancy management.

4. Code generation.

   Systems can be huge, and the order of several thousand of nodes is reached by practical applications. They are dynamic: critical systems are exposed to failures and mobility can be central in an application. *Simulation* is a key point in validation to measure latency delays, risks, or power consumption. Code generation integrate local behaviours inside a network topology, run the resulting simulation code, and provide diagnostics.

This is a compute intensive task where a number of steps must be executed by a number of nodes.

Further chapters will explain how to produce simulators for Graphic Accelerators, and for multi-threaded execution on multi-core processors, respectively from CUDA framework (x1000 processors), and Occam process oriented programs (x10 processors).

5. Preparing deployment.

   Once a system is validated, it is necessary to prepare an equivalent behavior for the sensors. This is also code generation, and can be achieved in a similar way as for simulators.

The following sections will describe existing functionality, and known challenges.

2.1 Network description

Network is described as a graph grouping nodes and communication links. In terms of data structures, a convenient representation is:

- a global Dictionary holding nodes,
- for each link, an input node, and an output node.
- for each node, a name, an array of input links, and an array of output links.

This representation allows to retrieve quickly the available nodes, or a particular node, and from that node, direct neighbors, by traversing each link.

2.1.1 Textual description

The textual representation is a reflect of this organization. It appears in the right editor of NetGen window:

- a title for the network,
- a list of messages carried by the links,
- one line for each node.

  These lines are a sequence grouping

  - the process name,
  - the neighborhood accessible by the output links, specified between braces, and separated by commas,
  - the name of the program, or procedure executed by the node,
  - node attributes

As an illustration, the network presented figure 1.6 has the following specification:

```
genRange100Points40

messages none  defined.
P1 { P11, P13, P23, P24 } Node (31 @ 182) (100)
P2 { P30 } Node (499 @ 40) (100)
P3 { P4, P6, P7, P33, P37 } Node (179 @ 338) (100)
P4 { P3, P7, P8, P31, P37 } Node (224 @ 269) (100)
P5 { P20 } Node (424 @ 306) (100)
P6 { P3, P7, P22, P33 } Node (227 @ 378) (100)
```

```
P7 { P3, P4, P6, P37 } Node (173 @ 316) (100)
P8 { P4, P22, P31, P33 } Node (293 @ 293) (100)
P9 { P10, P15, P21, P39 } Node (413 @ 601) (100)
P10 { P9, P15, P21, P39 } Node (410 @ 598) (100)
P11 { P1, P23 } Node (57 @ 112) (100)
P12 { P16, P22, P28 } Node (385 @ 440) (100)
P13 { P1, P23, P24, P37 } Node (89 @ 216) (100)
P14 { P32 } Node (269 @ 42) (100)
P15 { P9, P10, P21 } Node (350 @ 638) (100)
P16 { P12, P19, P22, P28 } Node (324 @ 448) (100)
P17 { P30, P32 } Node (368 @ 76) (100)
P18 { P38 } Node (153 @ 482) (100)
P19 { P16, P28 } Node (289 @ 513) (100)
P20 { P5 } Node (403 @ 283) (100)
P21 { P9, P10, P15, P28, P39 } Node (386 @ 558) (100)
P22 { P6, P8, P12, P16, P33 } Node (306 @ 386) (100)
P23 { P1, P11, P13, P37 } Node (108 @ 171) (100)
P24 { P1, P13, P35 } Node (18 @ 281) (100)
P25 { P29, P34, P36 } Node (580 @ 175) (100)
P28 { P12, P16, P19, P21 } Node (375 @ 487) (100)
P29 { P25, P34, P36 } Node (560 @ 152) (100)
P30 { P2, P17 } Node (420 @ 51) (100)
P31 { P4, P8 } Node (279 @ 237) (100)
P32 { P14, P17 } Node (281 @ 35) (100)
P33 { P3, P6, P8, P22 } Node (250 @ 380) (100)
P34 { P25, P29 } Node (537 @ 154) (100)
P35 { P24 } Node (41 @ 314) (100)
P36 { P25, P29 } Node (639 @ 172) (100)
P37 { P3, P4, P7, P13, P23 } Node (145 @ 255) (100)
P38 { P18 } Node (110 @ 436) (100)
P39 { P9, P10, P21 } Node (457 @ 570) (100)
```

2.1.2 Logic description

For small size networks, a logic organization can be processed by an external program called Graphviz. On Linux system, packages are available, thus on an Ubuntu system, it should be sufficient to load it (`apt-get install graphviz`). The input of this program is expressed in the *dot* syntax.

To produce dot files, select *Dot File* option which will produce a .dot file in the directory *Generated/*. When *Dot Graph* is selected, and where graphviz is available, the file is processed to produce a postscript representation that lot of viewers can read (see Figure).

2.1.3 Programming networks, and processing

The control window allows to save descriptions as .net text files, and to reload saved files. Processing these files can be done at any time by calling *accept* function in the editing facility. This will produce Dot files, Occam programs, CUDA programs when necessary.

The .net files can also be produced externally, specified within the editor, or the network structure can be produced by programs.

As a sample experiment a directional ring with 5 Nodes is specified as follows:

```
ring5

messages none  defined.
Head { P1  } Node
P1 { P2 } Node
P2 { P3 } Node
P3 { P4 } Node
P4 { Head } Node
```

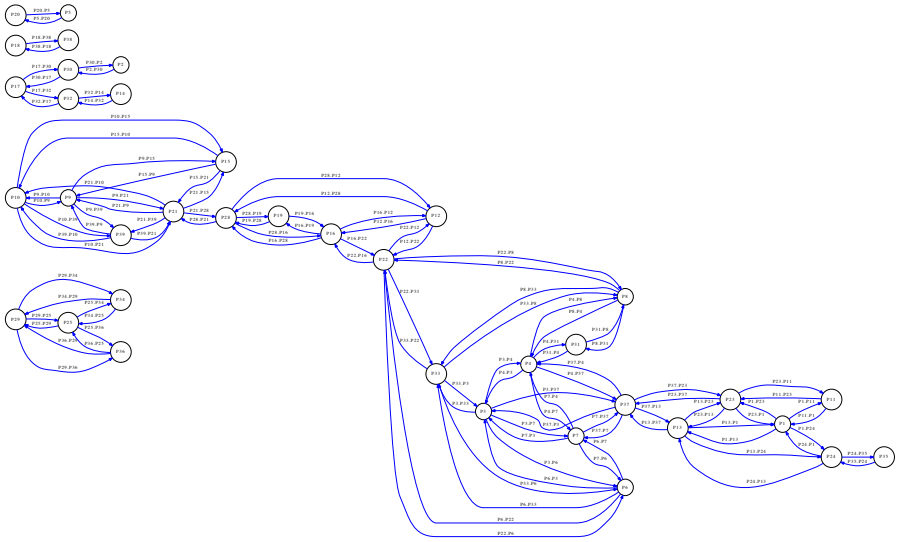

Figure 2.1: Logic organization as seen in *graphviz*

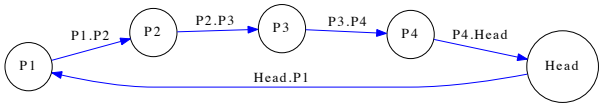

Figure 2.2: Ring5

2.1.4 Building networks by program

To be completed later.

The section can be skipped in a first stage. it implies some Smalltalk programming: one to two pages with pieces of listings.

2.2 Regular networks

To be completed later.

This function is called by selecting Grid rather than 2D Random, filling the range (200) and number of systems (40). The connectivity is computed on this basis..

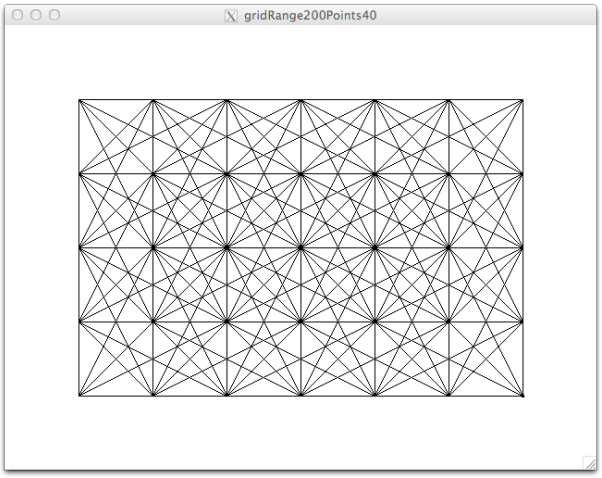

Figure 2.3: Layout on a rectangular surface with 200pts range and 40 nodes (at most).

2.3 Selecting a sensor layout from a map

Whereas the sensor network will be deployed, it is necessary to define sensing points, and expected connectivity between these points. Ranges produced by wireless technologies can be very different, very small for *body area networks* to very large, country size applications. Most of the solutions use dedicated network architectures that compute and route information, or standard solutions that support routing and sequencing of communications. In any case, the network topology is critical for two opposite reasons:

- reducing the number of communications is necessary to save energy and time,
- having enough redundancy in the routing capability is a solution in the case of failures (nodes or communications).

The frequency, volume and data rate of communications are also points of interest, with critical effects for some applications requiring high peek bandwidth. In other cases, frequency can be very low with the critical problems being energy and costs.

We will use a medium size geographic map example to illustrate network design, but anything else could apply (body description, nano fabric, etc...). Figure 2.4 is a PNG satellite view coming from the Internet, that also displays at the bottom left. An assumption is that a practical sensing application needs to deploy wireless equipment to measure some environmental characteristic. We also suppose that this equipment has been selected to

work on distances suitable to implement a network. As example, some 802,15,4 devices offer ranges from 20 to 40 km on the 900Mhz band.

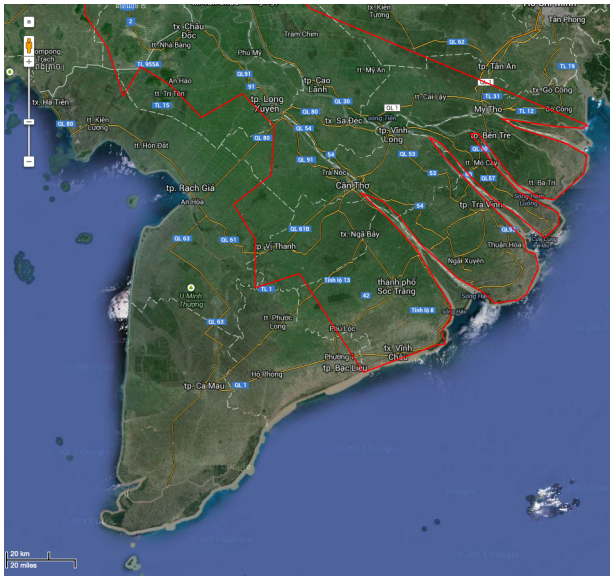

Figure 2.4: Sample image supposed to be stored as a PNG file in the working directory. The map has a scale useful to tune a range for wireless sensors.

2.3.1 Selecting sensor positions

In NetGen control window, use the *Options* > *Pick points* entry to open a new *Pick Points* window (figure 2.5). Then in this window, do a *File* > *Load image*, to load the sample image. The mouse cursor change to a cross, and each button pressed event will draw a circle around the selected position.

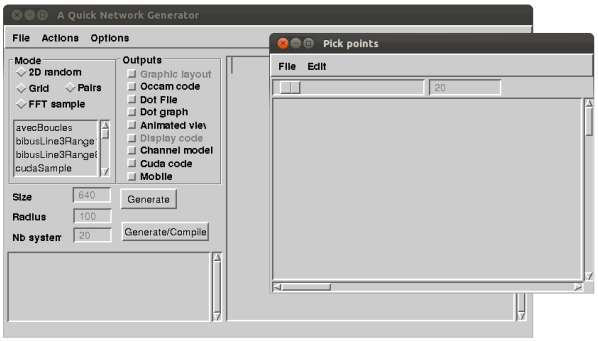

Figure 2.5: Pick point view

The slider on the top of the window, or the numeric field allow to change the range with the effect that circles around sensors increase, or decrease. when circles are large enough, sensors are supposed to establish radio contact ($distance(s1, s2) > range$).

A problem is to adapt the expected wireless range to the image, and a trick to do it is to install fake sensor points on the scale rule (shown at the bottom of figure 2.4), then to tune the slider to obtain a communication.

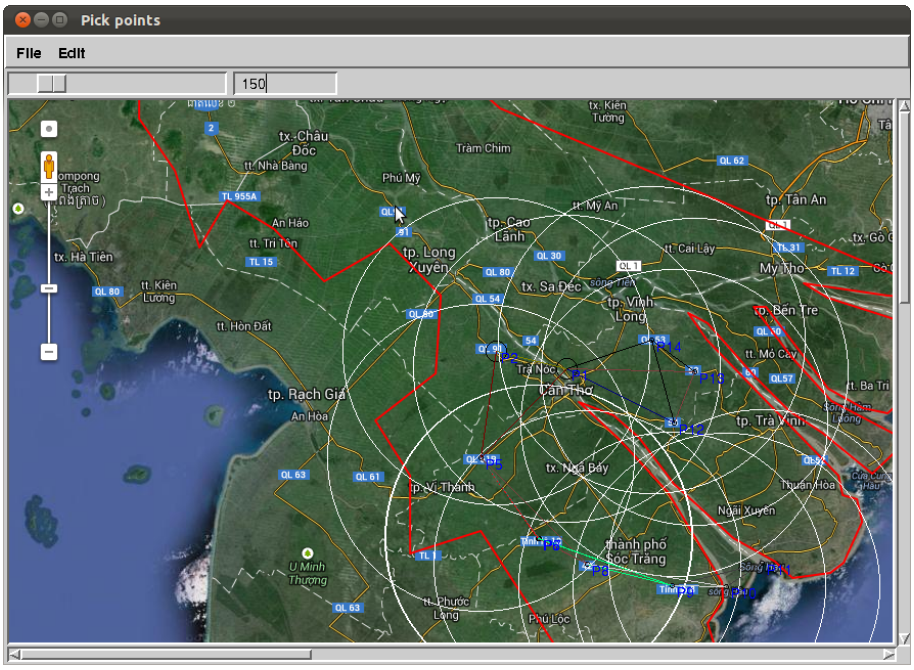

Figure 2.6: View showing sensor positions P_i), and connectivity. The scale has been adapted to 150 points for a distance of 20 km.

The *File* menu has options to save and reload points position into external text files.

2.3.2 Building a net

Still in the *File* menu, there is also a *Buildnet* entry, that presents the network specification inside the NetGen control window (see figure 2.7).

After transferring the specification, it is possible to edit it. As example it is a good idea to change its default name. It becomes also possible to use the code generation functions. Figure 2.8 presents a set of choice suitable for graphviz and Occam generation.

Notice that the call to these functions is done by the *accept* entry of a pop-up menu, *and not the Generate button* that will destroy the textual specification.

2.3.3 Logic presentation

The file has been dropped inside the Generated directory (section 2.1.2) as a Postscript file (figure 2.9). The *rule fake network* appears as a parasite on the left of the application network.

The logic file uses the same names as the *PickPoints* view, and the textual presentation.

2.4 Summary

This chapter explains how to use maps or other images for planing sensors positions, and how to check the topology by generating a graph view. The Generated directory also has an architecture description file expressed in the Occam Syntax.

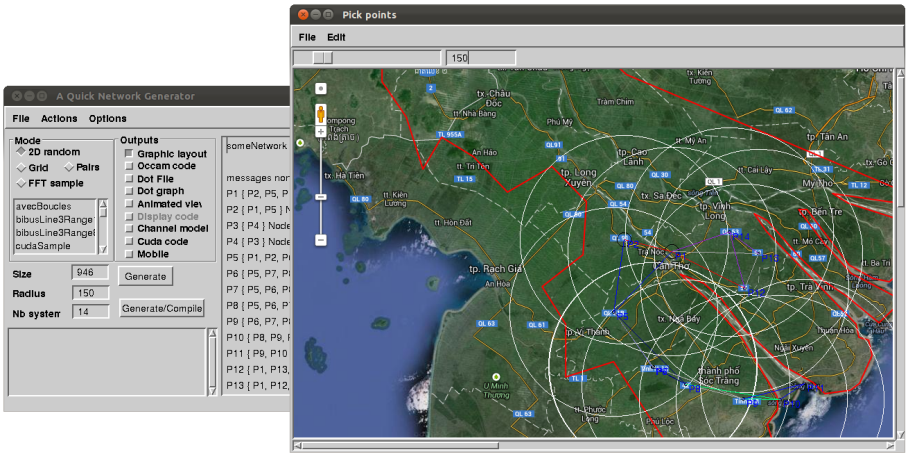

Figure 2.7: Transfer of the network to NetGen window.

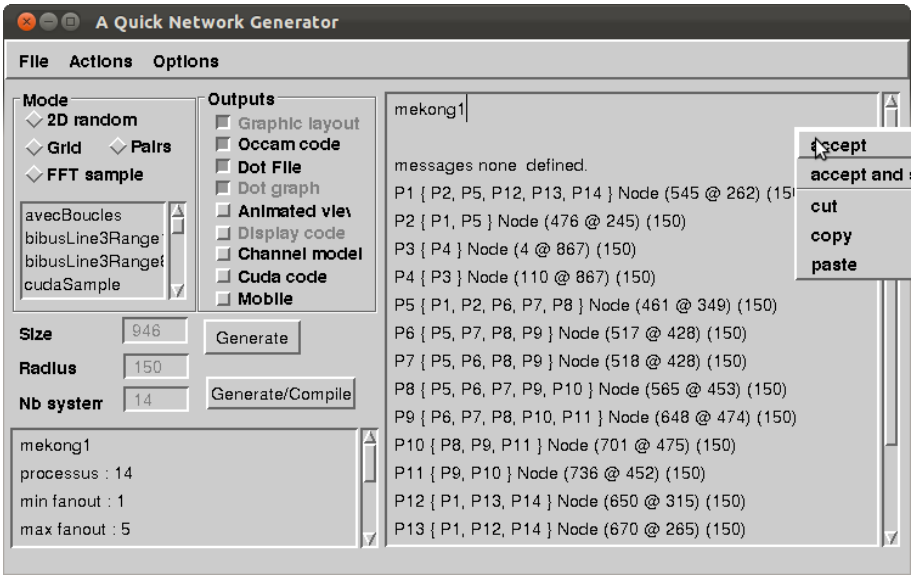

Figure 2.8: Generation of a logical network

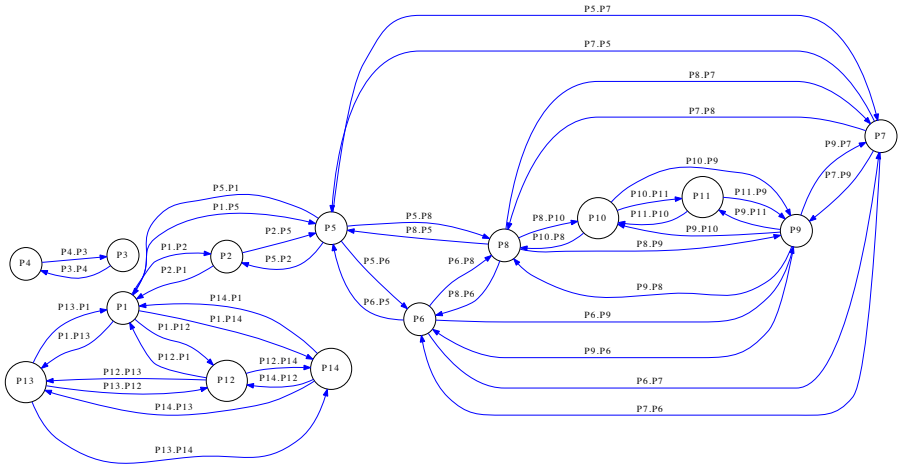

Figure 2.9: Equivalent logical network given a wireless range of 20 km

Chapter 3

Synchronous distributed behaviours using Occam

At this point, the installation of kroc, the *Kent Research Occam Compiler* should be considered. The first section will detail how to proceed on a Linux installation.

3.1 Installing kroc

The Occam compiler is developed at University of Kent, with its home page at `http://projects.cs.kent.ac.uk/projects`
Two branches are proposed: out of the same compiler frontend:

- kroc i386 compiler which makes use of a code generator for x86, thus enabling execution of programs on current multi-cores,
- Transterpreter Virtual Machine (TVM), which enables execution on micro-controllers.

The two branches are good targets for wireless sensor designs. The first is used to support concurrent simulation of networks, the second will support execution at sensor level in a portable way.

In this chapter, the i386 execution of simulations is the main concern, and our guide is the web page provided by Kent (it would be sufficient to follow these steps). This section is just a check out of instructions provided at this location.

1. access installation page from /sl Information for users
2. check your system

```
uname -a
Linux MyLinuxBox 3.2.0-51-generic #77-Ubuntu SMP Wed Jul 24 20:18:19 UTC 2013 x86_64 x86_64 x86_64 G
```

   I have a 64bits installation supporting concurrent execution.

```
lsb_release -a
LSB Version:    core-2.0-amd64:core-2.0-noarch:core-3.0-amd64:core-3.0-noarch:core-3.1-amd64:core-3.
Distributor ID: Ubuntu
Description:    Ubuntu 12.04.2 LTS
Release:        12.04
Codename:       precise
```

   This is Precise 12.04 LTS distribution of Ubuntu.

3. Fetch packages dependencies useful for kroc installation on Ubuntu as explained in Debian/Ubuntu (*Linux: If you're using Debian or Ubuntu, see DebianInstallation.*). For 32bits and 64bits installation, it is:

```
sudo apt-get install aptitude  bash gcc binutils gawk make automake autoconf pkg-config\
libc6-dev libsdl1.2-dev libsdl-sound1.2-dev libgl1-mesa-dev   libmysqlclient15-dev libpng12-dev
libxi-dev   libplayercore2-dev libplayerc2-dev libltdl3-dev   perl python xsltproc git
```

Additional step to support 32bits programs on 64bits systems such as MyLinuxBox:

```
sudo apt-get install libc6-dev-i386 lib32gcc1 gcc-multilib
```

And we can get back to the main installation page.

4. Fetching kroc sources using git (fast):

```
git clone --depth 1 -b kroc-1.6 git://github.com/concurrency/kroc.git kroc-git
```

This leaves a kroc-git additional directory with the sources. Change to this directory (`cd kroc-git`). There is a `build` command to configure the compiler sources, and, as mentioned in kroc web page one useful parameter would be to define the installation location.

5. Configuration and compilation of kroc *for end users* (we are end users), it takes time:

```
./build --prefix=/usr/local/kroc
```

On MyLinuxBox, we got errors, due to wrong installation of graphics libraries.

```
occbuild --in-tree /home/bernard/Documents/netgenDoc/kroc-git --toolchain=kroc --library occGL.li
\
-lglut -lGLU -lGL  -lSM -lICE  -lX11 -lXext -lXmu -lXt -lXi    opengl_wrap.o
/usr/bin/ld: cannot find -lglut
/usr/bin/ld: cannot find -lGLU
/usr/bin/ld: cannot find -lGL
/usr/bin/ld: cannot find -lSM
/usr/bin/ld: cannot find -lICE
...
```

A simple workaround is to start make with the ignore errors flag: `.make -i`

And the final diagnostic was :

```
KRoC has now been built.

Modules enabled (33/50):
  cif convert course dblmath dynproc file fmtout forall hostio hostsp http ioconv
maths occGL proc random raster rastergraphics rasterio rastertext selector shared_screen
snglmath sock solib splib ss stream string time trap ttyutil useful

Modules disabled (17/50):
  button cdx cspdrv graphics3d miniraster moa netbar occSDL occSDLsound occade occplayer ocuda pl
```

6. Now we install the programs in /usr/local/kroc, by typing : `sudo make -i install`.

Checking the installation we see a kroc compiler, and two shell scripts to configure the environment:

```
 ls /usr/local/kroc/bin
ilibr kmakef kroc kroc-setup.csh kroc-setup.sh mkoccdeps occ21 occamdoc occbuild tranx86 trapns
```

7. Obtain access to the compiler and checking access (bash version):

```
MyLinuxBox: $ source  /usr/local/kroc/bin/kroc-setup.sh
MyLinuxBox: $ which kroc
/usr/local/kroc/bin/kroc
MyLinuxBox: $ echo $LD_LIBRARY_PATH
/usr/local/kroc/lib
```

As this setup is to be done for each session, it is convenient to copy the script inside the shell configuration file (edit `/.bashrc`, as example).

And finally, we can launch the kroc compiler

```
MyLinuxBox: $  kroc
KRoC version 1.6.0 targeting x86_64-unknown-linux-gnu (driver V2.0)
Usage: kroc [options] [occam sources/pre-compiled sources]
Options:
  -b,  --brief           Give brief occ21 error messages
  -c,  --compile         Compile source to objects, do not link
  -s,  --strict          Strict checking mode
  -S,  --stoperror       Compile in STOP error mode
  -d                     Enable post-mortem debugging
  -di                    Enable insert debugging
  -e                     Enable user-defined channels
  -h,  --help            Print this message and exit
....
```

3.2 Checking Occam compiler: Hello world!

Samples to learn Occam programming are available under the examples directory for each module. Basic Occam examples are accessible in ./modules/course/examples and ./modules/course/exercises under the kroc-git directory.

```
MyLinuxBox: $ cat hello_seq_world.occ
PROC hello.world (CHAN BYTE keyboard?, screen!, error!)
  --{{{
  VAL []BYTE greeting IS "Hello World*c*n":
  SEQ i = 0 FOR SIZE greeting
    screen ! greeting[i]
  --}}}
:
```

- Build your own Occam directory : `mkdir /OccamDev`
- Copy example files there : `cp hello_seq_world.occ /OccamDev`
- Change to this directory : `cd /OccamDev`
- Compile by hand : `kroc hello_seq_world.occ`
- Execute : `./hello_seq_world`
  `Hello World`

Below is a commented version of the program. In Occam the program structure is defined by indentation of 2 spaces. This is visible for the body of the procedure, starting at VAL line, and for the loop, just below the SEQ statement.

```
-- start a comment
PROC hello.world (CHAN BYTE keyboard?, screen!, error!)
-- define a procedure named hello.world
-- with 3 communication links (channels carrying bytes)
-- associated to Linux i/o standard descriptors
  --{{{
-- this was an empty comment
  VAL []BYTE greeting IS "Hello World*c*n":
-- define a constant array of bytes with a string value, including CR
  SEQ i = 0 FOR SIZE greeting
-- sequential loop starting at i=0 with length of greeting occurrences
    screen ! greeting[i]
-- output a char to the screen channel
  --}}}
:
-- end of procedure mark.
```

3.3 Parallel construct and channels in Occam

Coming back to the topic of network simulation, this section will construct a concurrent program suitable for the directional ring displayed figure 2.2. Each node in the ring could represent a sensor. Sensors common behaviour is to execute an infinite loop for:

1. sensing, loading some status variables with values observed locally,
2. communications
   (a) sending information to direct neighbors,
   (b) receiving information from neighbors,
3. sleeping for an agreed fixed period

3.3.1 Sample ring5 behaviour

Let us start our example as a very simple program. Each sensor activity is represented by a process, and each process executes the same program, defined as a procedure `Node.v1`. Communication links are represented by Occam channels carrying integers. To distinguish sensor from each other, it is necessary to provide a unique identifier `Identity`.

Then, as sensing is supposed to produce some result in a local variable `Local.Value`, we will simply increment this variable.

To communicate, we pass the variable to one of our next neighbor, and receive the value from our previous neighbor.

This *behaviour* is programmed in a `ring5.v1.occ` file as follows:

```
PROC Node.v1 (CHAN OF INT Incoming.Chan,Outgoing.Chan, VAL INT Identity)
  INT Local.Value, Incoming.Value :
  SEQ
    Local.Value := Identity
    WHILE TRUE
      SEQ
        Local.Value := Local.Value +1 -- 1 sensing
        PAR -- 2 communication
          Outgoing.Chan ! Local.Value
          Incoming.Chan ? Incoming.Value
        SKIP -- 3 sleeping
:
```

Notice that step 2 is programmed with a PAR construct over sending and receiving. We don't want to define an order for activities that are concurrent. Furthermore, programming sequential communications would lead to a dead-lock in the simulated ring, Occam channels being blocking channels: communication is resolved as the 2 processes reach a synchronization point. The concurrent construct finishes with the last branch, as a *barrier* condition.

To check the grammatical correctness of this program, we can add an empty main activity, just after the `Node.v1` procedure definition:

```
PROC Sys(CHAN OF BYTE in,out,err)
  SEQ
    SKIP
:
```

Then, we compile our file `ring5.v1.occ`, and we execute the result:

```
MyLinuxBox: $ kroc ring5.v1.occ
Warning-occ21-ring5.v1.occ(17)- parameter err is not used
Warning-occ21-ring5.v1.occ(17)- parameter out is not used
Warning-occ21-ring5.v1.occ(17)- parameter in is not used
MyLinuxBox: $  ./ring5.v1
MyLinuxBox: $
```

This programs does nothing since the SKIP statement denotes an empty process.

3.3.2 Sample ring5 architecture

To obtain a more convincing ring, we need to define a ring architecture having 5 nodes, and 5 communication links. This is done by replacing the the Sys definition by a more complete one inside a new file `ring5.v2.occ`.

```
PROC Sys(CHAN OF BYTE in,out,err)
  -- channels definition
  CHAN OF INT P1.P2, P2.P3, P3.P4, P4.P5, P5.P1:
  -- concurrent ring construct
  PAR
    Node.v1 (P5.P1,P1.P2,1)  -- P1
    Node.v1 (P1.P2,P2.P3,2)  -- P2
    Node.v1 (P2.P3,P3.P4,3)  -- P3
    Node.v1 (P3.P4,P4.P5,4)  -- P4
    Node.v1 (P4.P5,P5.P1,5)  -- P5
:
```

Now we compile and execute. This will produce a program with an infinite loop to be killed. Notice that each channel is used 2 times, in input and output parameter positions. Kroc check correctness of the architecture with two user process for reading and writing.

```
MyLinuxBox: $ kroc ring5.v2.occ
Warning-occ21-ring5.v2.occ(19)- parameter err is not used
Warning-occ21-ring5.v2.occ(19)- parameter out is not used
Warning-occ21-ring5.v2.occ(19)- parameter in is not used
MyLinuxBox: $ ./ring5.v2
```

3.3.3 Ring 5 has a synchronous behaviour

Each process has its own control loop, but the PAR communication implementation guarantees that none of them can take much progress against the neighbors. Every process is in the same turn as the other ones.

The simulation is executed under Occam micro-kernel controller called CCSP, that can be multi threaded and distributed on several processor cores. The behaviour is semantically equivalent to what happens in a wireless network whatever is the protocol used in the MAC layer (time division TDMA, CSMA, acknowledged or not).

This simulation also obeys to synchronous distributed algorithms methodology, that bring lots of opportunities for defining how the sensor network will implement services and overcome difficulties.

3.4 Observing execution, simulation traces

The program in section 3.3.2 does not produce any usable output. To allow observation of its behaviour, we need some external print out on what is happening.

Unfortunately, printing in text on a terminal requires sharing the terminal, thus synchronization of processes willing to print. This can be overcome with graphics presentation, but let us see what we can do about sharing i/os.

We have seen section that Occam program have channels mapped on file descriptors. In the case of Ring5, the `stdout` descriptor must be written by our 5 processes. This is achieved by a multiplexer, and there is the ALT construct of Occam that allows to take into account 5 channels selecting one of them which appears to be ready.

ALT has an entry for channel to be inspected. A ready channel value is read in a variable, an after this, an action is taken. As an example let us send two channels `c.in.1` `c.in.2` into one channel `c.out` :

```
CHAN OF BYTE c.in.1, c.in.2, c.out:
BYTE char:
SEQ i=0 FOR MaxTurns
```

```
ALT
  c.in.1 ? char
    c.out! char
  c.in.2 ? char
    c.out! char
```

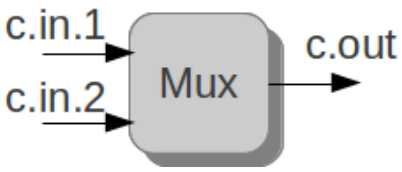

Figure 3.1: Multiplexer on 2 channels

Normally, the construct is non determinist, meaning that one channel is selected at random. Also, only one of the ready channels is taken for each resolved ALT, and the construct block until one of its entry is ready.

3.4.1 Programming a trace multiplexer

The procedure Mux below shows a construct for a fixed number of nodes (MaxNodes) looping synchronously MaxTurns times.

```
#USE "course.lib"
-- enables to use formatted printing procedures, out.number(..)
VAL INT MaxNodes IS 5:
-- we have 5 nodes

VAL INT MaxTurns IS 10:
-- we will do 10 rounds

-- Mux is our observer in the system
PROC Mux([]CHAN OF BYTE muxTab, CHAN OF BYTE out)
-- muxTab is an array of input channels
-- its size is managed by the Occam compiler
-- out is the output channel
  BYTE char:
-- input char
  SEQ i=0 FOR  (MaxNodes * MaxTurns)
    ALT i=0 FOR SIZE muxTab
-- fetch the real size of the array
      muxTab[i] ? char
-- block until one of the input is ready
-- i is the index of the selected ready channel
        SEQ
          out.number(i,4,out)
          -- print the index of the channel
           out ! '*t'
          -- print a tab
         out ! char
          -- print the char
          WHILE char <> '*n'
          -- loop to the end of the line
            SEQ
              muxTab[i] ? char
              out ! char
              -- read char on the channel and print it
:
```

This code is suitable to trace MaxNodes nodes, each of them writing on an entry of a table, a full line closed by an end of line.

3.4.2 Ring behavior with a trace

As we want to watch what is happening in each process, we need to add a channel to the Mux into each process, and to use this channel inside the internal loop. As we have restricted the Mux to MaxTurns rounds, we also need to exchange the infinite loop to a restricted sequence. This is the modified Node.v2 procedure:

```
PROC Node.v2 (CHAN OF INT Incoming.Chan,Outgoing.Chan, VAL INT Identity, CHAN OF BYTE To.Mux)
  INT Local.Value, Incoming.Value :
  SEQ
    Local.Value := Identity
    WHILE TRUE
      SEQ
        Local.Value := Local.Value +1 -- 1 sensing
        PAR -- 2 communication
          Outgoing.Chan ! Local.Value
          Incoming.Chan ? Incoming.Value
        SKIP -- 3 sleeping
        out.number(Local.Value,0,To.Mux)
        To.Mux ! '*n'
        -- trace the value of the local variable and send CR
:
```

3.4.3 Ring architecture with trace multiplexer

Now we implement the full program with:

1. Mux procedure as shown section 3.4.1, then
2. Node procedure from section 3.4.2

And we need to complete the process system from section 3.3.2 by declaring channels from processes to the trace collector, and add these channels in the parallel construct branches. It is also needed to call the Mux with its array of input channels and the system stdout access (see figure 3.2):

```
PROC Sys(CHAN OF BYTE in,out,err)
  -- channels definition
  CHAN OF INT P1.P2, P2.P3, P3.P4, P4.P5, P5.P1:
  [MaxNodes] CHAN OF BYTE To.Mux.Tab:
  -- concurrent ring construct
  PAR
    Mux(To.Mux.Tab,out)
    Node.v2 (P5.P1,P1.P2,1,To.Mux.Tab[1-1])  -- P1
    Node.v2 (P1.P2,P2.P3,2,To.Mux.Tab[2-1])  -- P2
    Node.v2 (P2.P3,P3.P4,3,To.Mux.Tab[3-1])  -- P3
    Node.v2 (P3.P4,P4.P5,4,To.Mux.Tab[4-1])  -- P4
    Node.v2 (P4.P5,P5.P1,5,To.Mux.Tab[5-1])  -- P5
:
```

The source can be compiled asking a link with the course library, then executed filtering the 10 first lines.

```
MyLinuxBox $ kroc -lcourse ring5.v3.occ
Warning-occ21-ring5.v3.occ(53)- parameter err is not used
Warning-occ21-ring5.v3.occ(53)- parameter in is not used
MyLinuxBox $ ./ring5.v3 | head -10
   4    6
   0    2
   1    3
   2    4
   0    3
   1    4
   3    5
   4    7
   2    5
   0    4
```

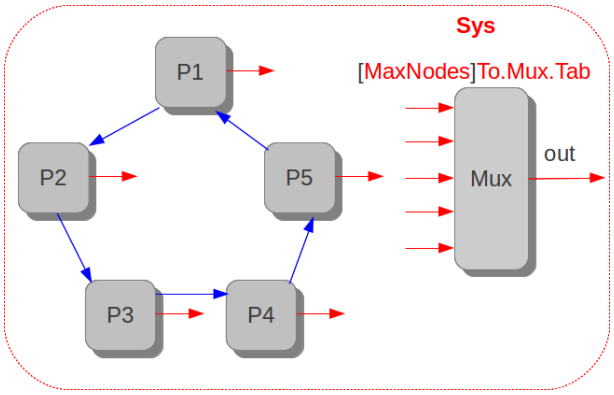

Figure 3.2: Complete Ring5 architecture with trace multiplexer

If we want to see the sequence of numbers output by node 1, we can use grep to filter this node as well:

```
MyLinuxBox $  ./ring5.v3 | grep '^ *1'
   1     3
   1     4
   1     5
   1     6
   1     7
   1     8
   1     9
   1     10
   1     11
   1     12
```

3.5 Architectures and Behaviors in NetGen framework

In section 2.1.3 we have shown how to produce graphs, and Occam description too. It is time to come back to the initial ring example and to watch what comes out from the generator. The files produced describes architectures in a generic way. Whatever is the network, we should be able to run our distributed behavior on it. The `Generated/` directory contains the ring architecture in the `ring5.occ` file. Let us comment its contents in a simplified way.

3.5.1 Occam architecture description from NetGen

There are 2 major differences with code detailed in section :

- the program has been split in 2 files, one for architecture `ring5.occ`, and one for behavior `nodes-test-include.occ`. The first one (generated) includes the second one (written by hand).

- instead of listing all the channels in procedure parameters, we group them into tables, and pass these tables as parameters. PROC Ring5 thus define input and output group of channels, and pass them when starting the process:

```
Head.out IS [ Head.P1 ]:
Head.in IS [ P4.Head ]:
-- and later
Node(Head.in, Head.out,0, toMux [0])
```

A big advantage in doing this is that we can have different connectivity for different processes, and the connectivity can become very large.

This will be demonstrated later on large network examples.

```
#USE "course.lib"  -- support for printing
VAL INT MaxFanOut IS 1: --max number of channels per node
VAL INT MaxNodes IS 5: -- max number of nodes

#INCLUDE "nodes-test-include.occ"
-- includes the file where the behaviour is located
-- this file must contains definitions for procedures Node and Mux
-- plus the diam.proto type for communication links

PROC ring5(CHAN OF BYTE stdin, stdout, stderr)
   -- Channel declarations
  CHAN OF diam.proto Head.P1:
  CHAN OF diam.proto P1.P2:
  CHAN OF diam.proto P2.P3:
  CHAN OF diam.proto P3.P4:
  CHAN OF diam.proto P4.Head:

  -- Channel table declaration for nodes
  Head.out IS [ Head.P1 ]:
  Head.in IS [ P4.Head ]:
  P1.out IS [ P1.P2 ]:
  P1.in IS [ Head.P1 ]:
  P2.out IS [ P2.P3 ]:
  P2.in IS [ P1.P2 ]:
  P3.out IS [ P3.P4 ]:
  P3.in IS [ P2.P3 ]:
  P4.out IS [ P4.Head ]:
  P4.in IS [ P3.P4 ]:

  -- Program Body
  [MaxNodes]CHAN OF BYTE toMux:
  PAR
    Node(Head.in, Head.out,0, toMux [0])
    Node(P1.in, P1.out,1, toMux [1])
    Node(P2.in, P2.out,2, toMux [2])
    Node(P3.in, P3.out,3, toMux [3])
    Node(P4.in, P4.out,4, toMux [4])
    Mux(toMux,stdout)
     -- End of program body
:
```

3.5.2 Behaviour description, first approach

We know copy our previous behavior in a `nodes-test-include.occ`, and noticing that we receive array of channels, we modify the Node procedure, using the first entry of these arrays.

It is also necessary to declare a `diam.proto` type as being an INT, and to edit the Node procedure with cast and correct declaration of variables.

This is a first version of the behavior file:

```
DATA TYPE diam.proto IS INT:
VAL INT MaxTurns IS 10:

PROC Mux([]CHAN OF BYTE muxTab, CHAN OF BYTE out)
  BYTE char:
  SEQ i=0 FOR  (MaxNodes * MaxTurns)
    ALT i=0 FOR SIZE muxTab
      muxTab[i] ? char
        SEQ
          out.number(i,4,out)
```

```
          out ! '*t'
          out ! char
          WHILE char <> '*n'
            SEQ
              muxTab[i] ? char
              out ! char
:

PROC Node([]CHAN OF diam.proto Incoming.Chan,Outgoing.Chan, VAL INT Identity, CHAN OF BYTE To.Mux)
  INT Local.Value:
  diam.proto Incoming.Value:
  SEQ
    Local.Value := Identity
    SEQ i=0 FOR MaxTurns
      SEQ
        Local.Value := Local.Value +1 -- 1 sensing
        PAR -- 2 communication
          Outgoing.Chan[0] ! (diam.proto Local.Value)
          Incoming.Chan[0] ? Incoming.Value
        SKIP -- 3 sleeping
        out.number(Local.Value,0,To.Mux)
        To.Mux ! '*n'
        -- trace the value of the local variable
:
```

The file to compile is the architecture, the execution produces the same result as in section 3.5.1.

```
MyLinuxBox $ ls
nodes-test-include.occ  ring5.occ
MyLinuxBox $ grep INC  ring5.occ
#INCLUDE "nodes-test-include.occ"
bernard@PedelBP:~/Documents/netgenDoc/Ring5$ kroc -lcourse ring5.occ
Warning-occ21-ring5.occ(34)- parameter stderr is not used
Warning-occ21-ring5.occ(34)- parameter stdin is not used
MyLinuxBox $ ./ring5 | head -8
   4    5
   0    1
   1    2
   2    3
   0    2
   1    3
   3    4
   4    6
```

3.6 Summary : flow for generated bidirectional ring

Let us review specification and architecture code generation on the case of a bidirectional 4 nodes ring. We will need to modify the behavioral part of the program, and will be ready for final statements on using code generation for any network.

3.6.1 Specification and drawing

This network will be called BiDirRing4. It is processed in the same way as section 2.1.3, asking for Occam generation and graphviz generation.

```
BiDirRing4
messages none  defined.
P1 { P2, P4 } Node
P2 { P3, P1 } Node
P3 { P4, P2  } Node
P4 { P1, P3  } Node
```

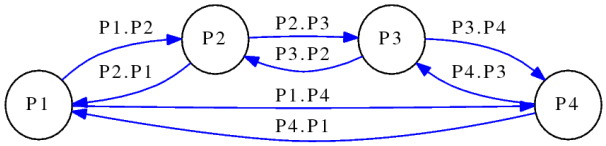

Figure 3.3: Logic organization for 4 nodes Bi-directional ring

3.6.2 Occam resulting architecture

The Generated directory contains the architecture description `BiDirRing4.occ` from which is extracted the code below (shortened). One can notice that channel arrays now contain 2 links rather than one (`P1.out IS [ P1.P2,P1.P4 ]:`).

```
-- processus : 4
-- min fanout : 2
-- max fanout : 2
-- channels   : 8

#USE "course.lib"


VAL INT MaxFanOut IS 2:
VAL INT MaxNodes IS 4:

#INCLUDE "nodes-test-include.occ"

PROC BiDirRing4(CHAN OF BYTE stdin, stdout, stderr)
   -- Channel declarations
  CHAN OF diam.proto P1.P2,P1.P4:
  CHAN OF diam.proto P2.P3,P2.P1:
  CHAN OF diam.proto P3.P4,P3.P2:
  CHAN OF diam.proto P4.P1,P4.P3:

  -- Channel table declaration for nodes
  P1.out IS [ P1.P2,P1.P4 ]:
  P1.in IS [ P2.P1,P4.P1 ]:
  P2.out IS [ P2.P3,P2.P1 ]:
  P2.in IS [ P1.P2,P3.P2 ]:
  P3.out IS [ P3.P4,P3.P2 ]:
  P3.in IS [ P2.P3,P4.P3 ]:
  P4.out IS [ P4.P1,P4.P3 ]:
  P4.in IS [ P1.P4,P3.P4 ]:

  -- Program Body
  [MaxNodes]CHAN OF BYTE toMux:
  PAR
    Node(P1.in, P1.out,0, toMux [0])
    Node(P2.in, P2.out,1, toMux [1])
    Node(P3.in, P3.out,2, toMux [2])
    Node(P4.in, P4.out,3, toMux [3])
    Mux(toMux,stdout)
     -- End of program body
:
```

3.6.3 General formulation for behavior

The need is to show how to read and write several channels rather than one. To allow this, it is needed to provide as many buffers as there are input and output links. The maximum connectivity in the network is known in a constant MaxFanOut. Thus, we can dimension and control these buffers.

As we are sending from buffers, it is also necessary to copy state values, or produce messages in the buffers, and similarly, we will need to collect and examine incoming messages to update node status.

Data type diam.proto, and procedure Mux does not change. An updated Node procedure appears as follows in a new version of `nodes-test-include.occ`:

```
PROC Node([]CHAN OF diam.proto Incoming.Chan,Outgoing.Chan, VAL INT Identity, CHAN OF BYTE To.Mux)
  [MaxFanOut] INT Local.Values:  -- buffers for outgoing messages
  [MaxFanOut] diam.proto Incoming.Value: -- buffers for incoming
  INT Local.Value:
  SEQ
    Local.Value := Identity
    SEQ i=0 FOR MaxTurns
      SEQ
        Local.Value := Local.Value +1 -- 1 sensing
        SEQ i=0 FOR MaxFanOut -- copy our state to outgoing buffers
          Local.Values[i]:= Local.Value
        PAR -- 2 communication
          PAR index = 0 FOR MaxFanOut -- send in parallel
            Outgoing.Chan[index] ! (diam.proto Local.Values[index])
          PAR index = 0 FOR MaxFanOut -- receive in parallel
            Incoming.Chan[index] ? Incoming.Value[index]
        out.number(Local.Value,0,To.Mux) -- trace some state
        To.Mux ! '*n'
:
```

Compile and execute in a specific directory `BiDirRing4`:

```
MyLinuxBox $ ls
BiDirRing4  BiDirRing4.occ  nodes-test-include.occ
MyLinuxBox $ kroc -lcourse BiDirRing4.occ
Warning-occ21-BiDirRing4.occ(32)- parameter stderr is not used
Warning-occ21-BiDirRing4.occ(32)- parameter stdin is not used
MyLinuxBox $ ./BiDirRing4 | head -6
   3    4
   0    1
   1    2
   2    3
   0    2
   1    3
```

3.6.4 Exercise

Verify that you can produce a trace for an 8 nodes bidirectional ring for the same behaviour..

3.6.5 Exercise

BiDirRing4 is not a good demonstration of cooperation between nodes since the program ignores values in incoming messages. A step forward would be to compute mean of a value distributed in the neighborhood:

- modify BiDirRing4 to send a local value to the direct neigbors
- receive values from the neigbors and compute the mean of these values including the local one.
- repeat the process for neighborhoods of 5 nodes inside the ring. Do a trace analysis.

Chapter 4

Distributed algorithms simulation

Be fore developing actual programs for wireless sensor networks, it is good to check if the cooperation of local programs will lead to working and efficient results.

The distributed behaviour comes from:

- an architecture specification, such as `mekong1.occ`,
- a behaviour executed by nodes, such as `nodes-test-include.occ`.

The two descriptions are orthogonal, meaning that one can make evolution on the architecture at fixed behaviour, or make evolutions on the behaviour with fixed architecture. The situation is well known in computer architecture. It is named an Y methodological approach, and was popularized by Gajski. The bottom branch carrying measures produced from tools (energy, response time, cost, etc. . .).

Contents