Dynamic networks

NetGen : objectives, installation, use, and programming

Bernard Pottier
Pierre-Yves Lucas, . . .
Université de Bretagne Occidentale

January 29, 2015

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.

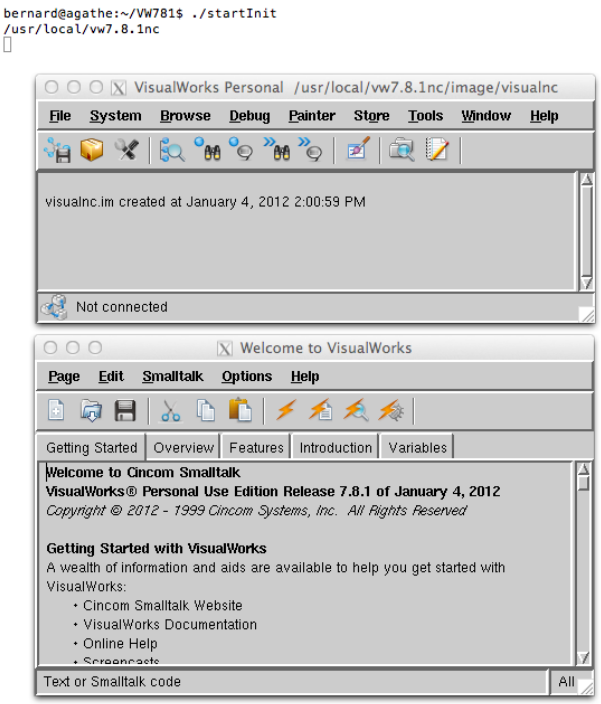

Figure 1.1: Initial environment

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.

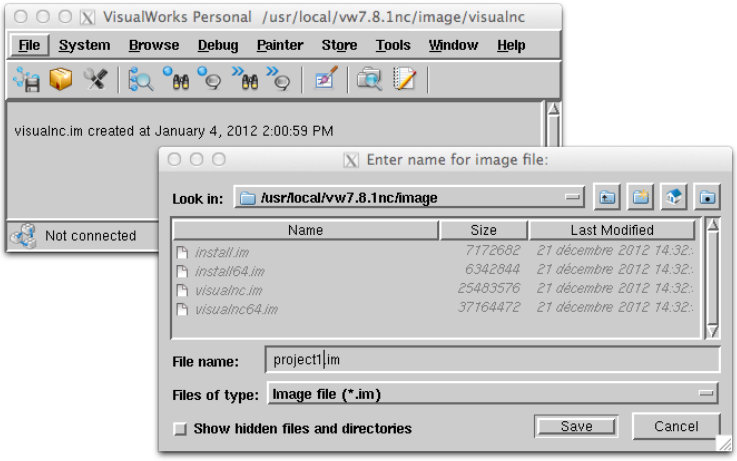

Figure 1.2: Creating a new image file *project1.im*

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.

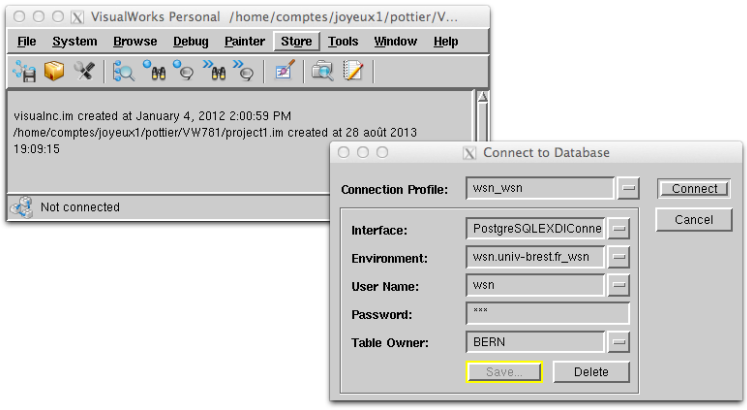

Figure 1.3: Defining a repository access

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.

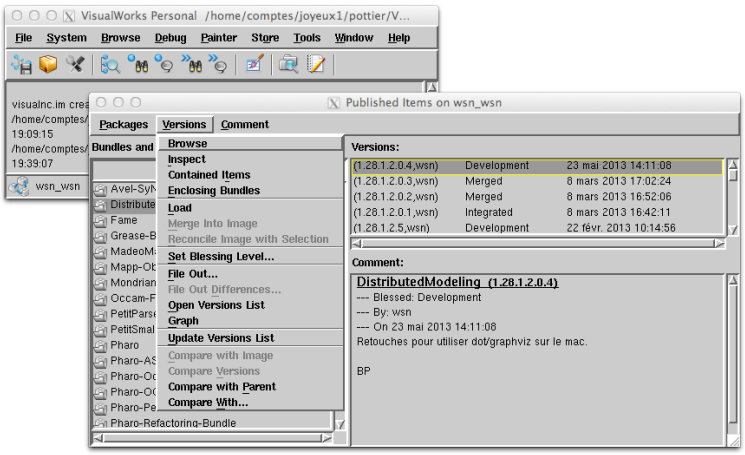

Figure 1.4: Choosing and loading packages

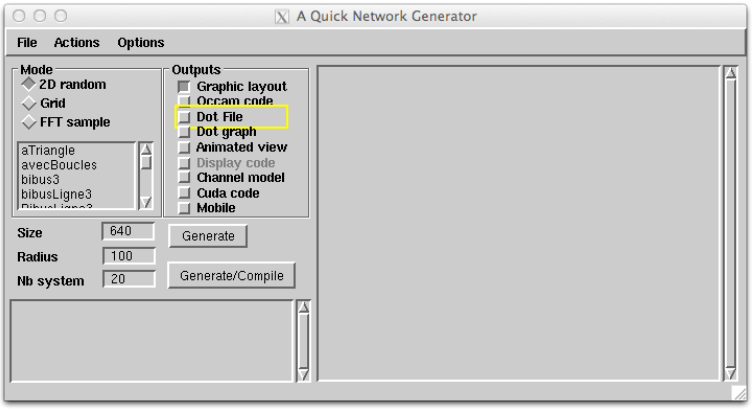

Figure 1.5: NetGen initial window

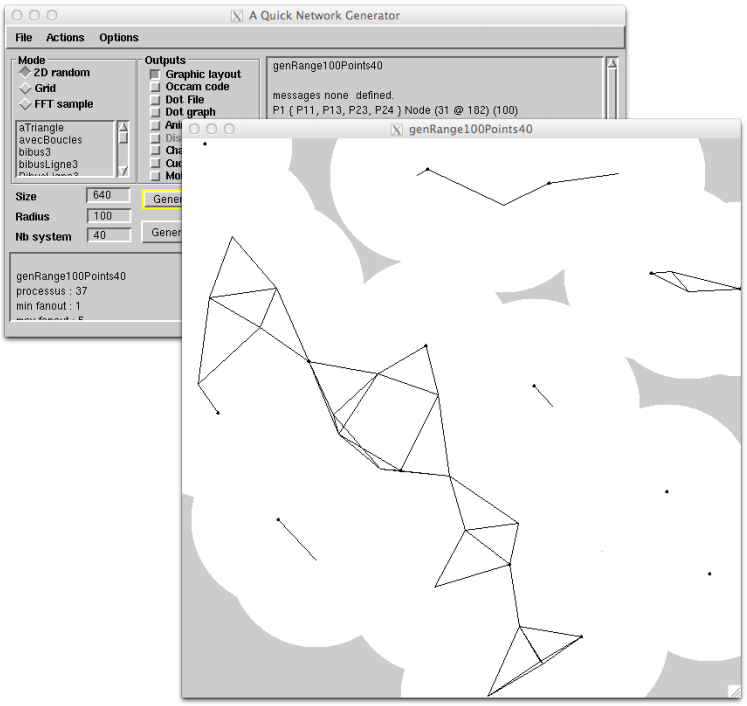

Figure 1.6: NetGen : random network generation with 40 nodes

```
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.

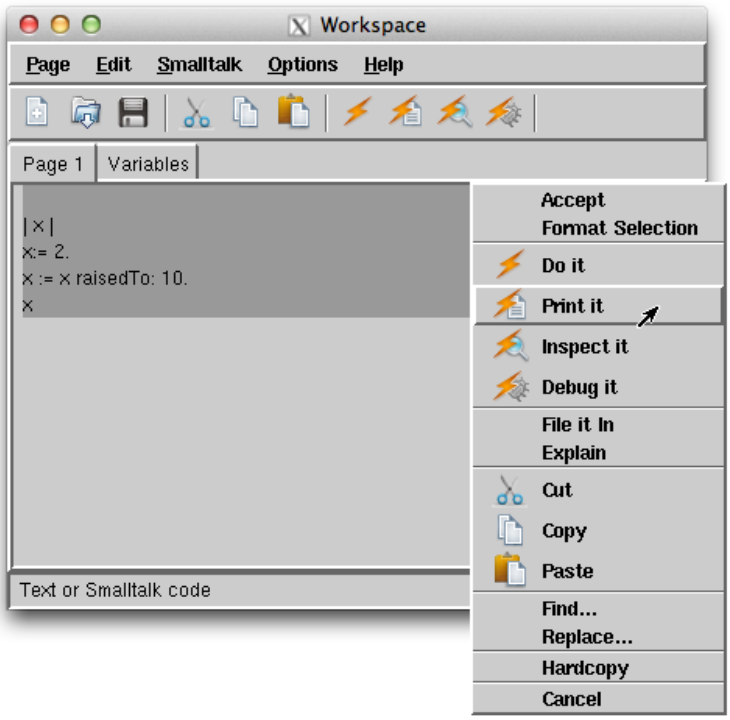

Figure 1.7: Programming in work-spaces

- 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
```

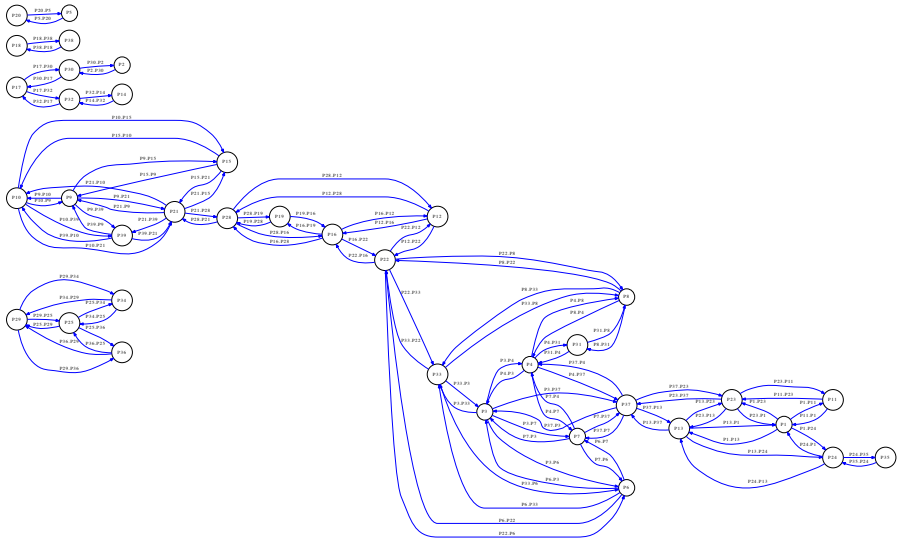

Figure 2.1: Logic organization as seen in *graphviz*

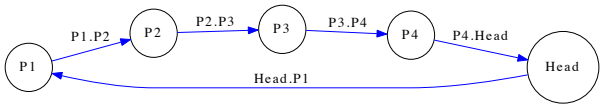

Figure 2.2: Ring5

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..

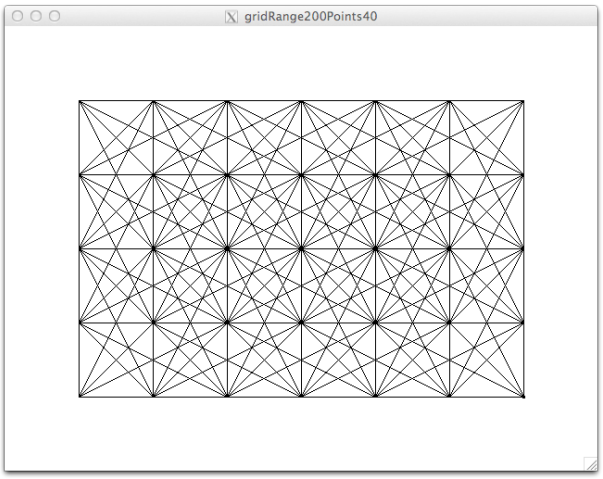

Figure 2.3: Layout on a rectangular surface with 200pts range and 40 nodes (at most).

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.

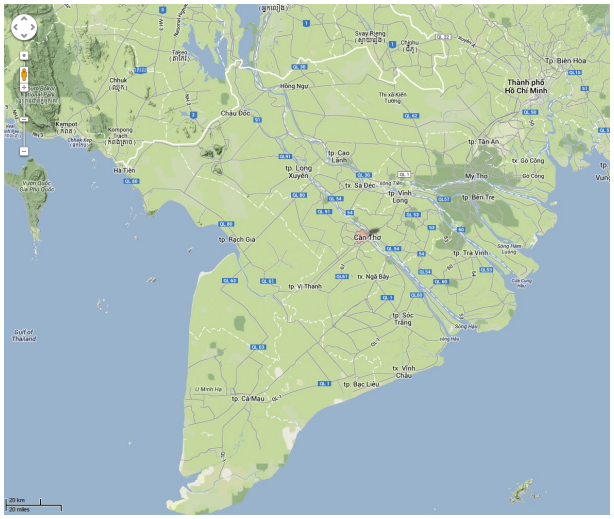

Figure 2.4: Sample image supposed to be stored as a PNG file in the working directory. The map has a scale useful to tune a range for wireless sensors.

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.

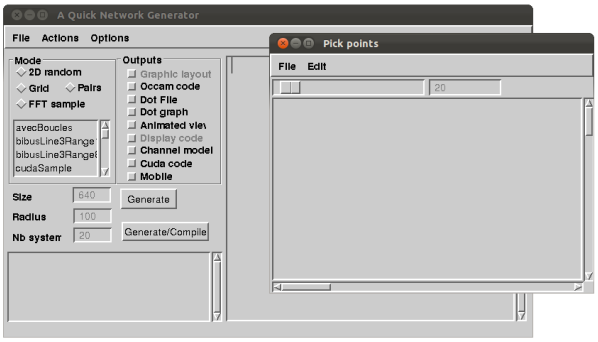

Figure 2.5: Pick point view

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.

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

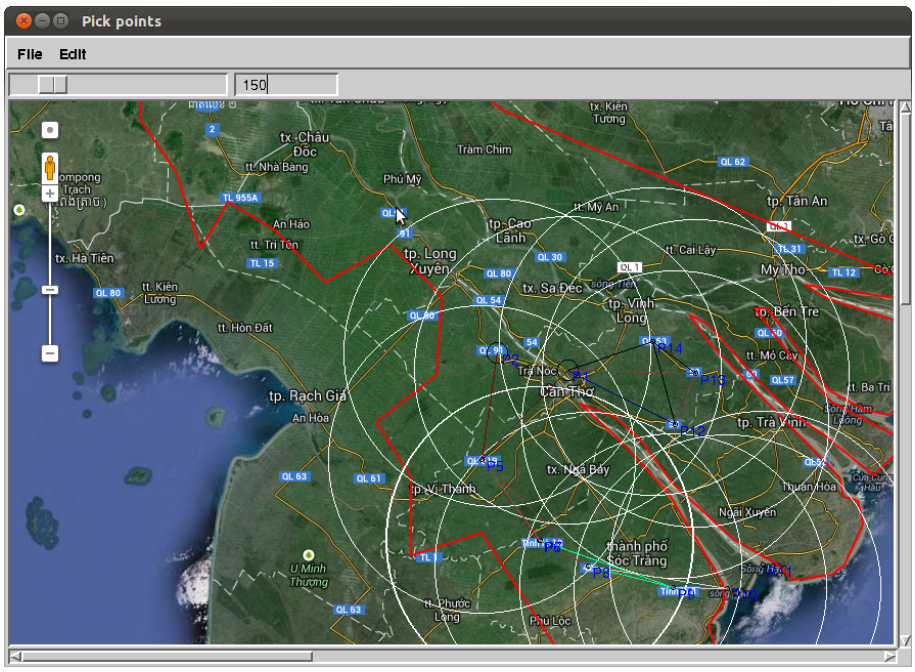

Figure 2.6: View showing sensor positions P_i), and connectivity. The scale has been adapted to 150 points for a distance of 20 km.

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.

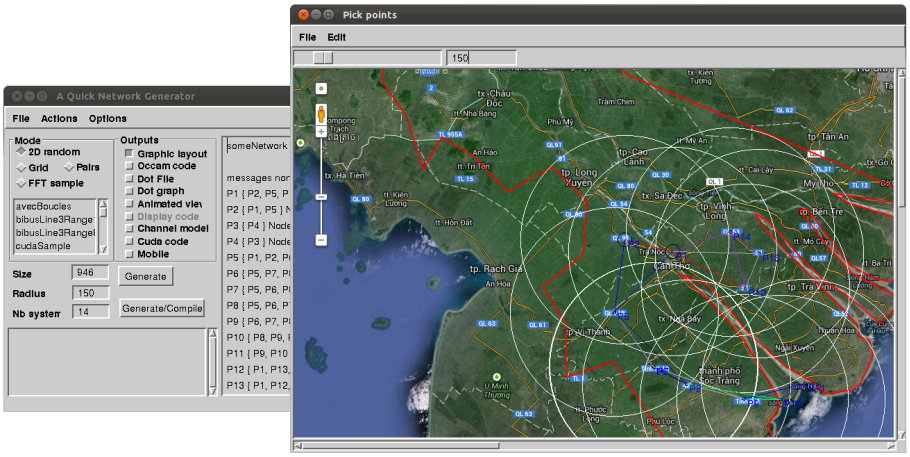

Figure 2.7: Transfer of the network to NetGen window.

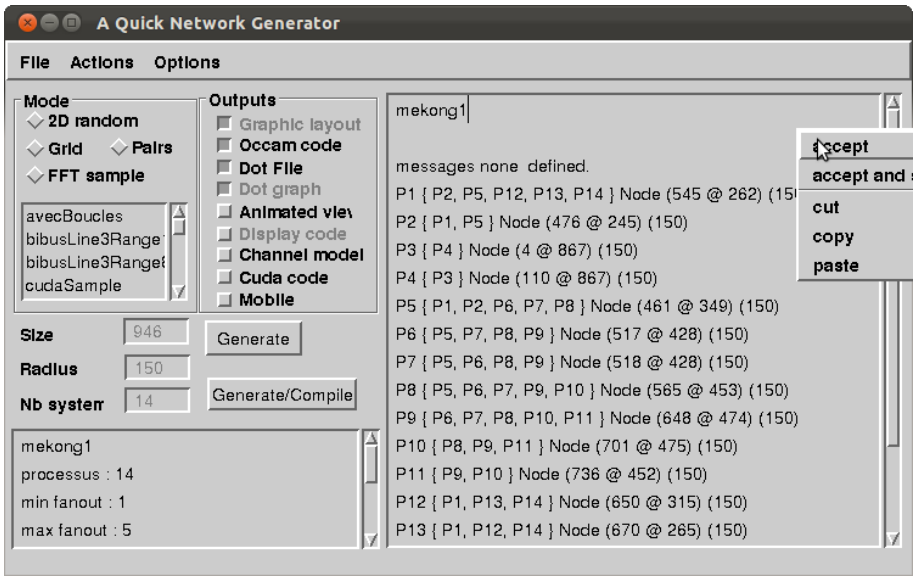

Figure 2.8: Generation of a logical network

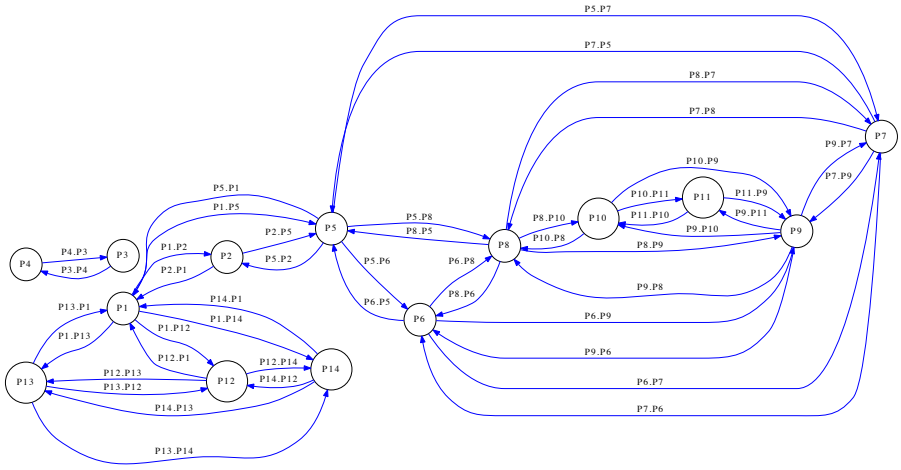

Figure 2.9: Equivalent logical network given a wireless range of 20 km

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/proje`
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_
```

   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:cor
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-d
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 occG
\   ||
-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 ocud
```

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 trap
```

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
```

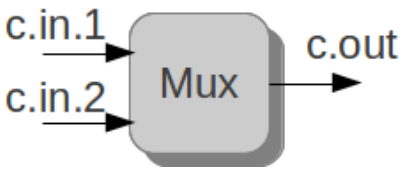

Figure 3.1: Multiplexer on 2 channels

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
```

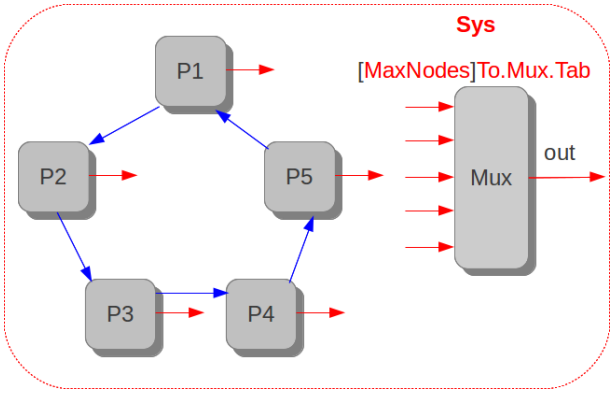

Figure 3.2: Complete Ring5 architecture with trace multiplexer

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.M
  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
```

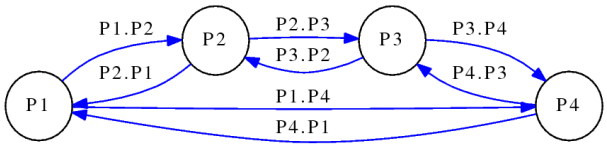

Figure 3.3: Logic organization for 4 nodes Bi-directional ring

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.M
  [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...).

Simulation is a key method to take measures on complex situations. To use simultation, we need to reproduce real dispersion of behaviours, and random number generators are useful for this.

4.1 Random numbers in Occam

The Occam `course` library provides the random function that has two parameters and provides a result as a couple of numbers (integers).

The small program randomSample.occ demonstrates the general use for random.

```
#USE "course.lib"
-- montre le fonctionnement du generateur aleatoire.
PROC Random(CHAN OF BYTE in,out,err)
  VAL INT N IS 5:
  VAL INT K IS 10:  -- borne de tirage de random : [0,K[
  PROC Genere(VAL INT seedInitial)
    INT x,seed:
    SEQ
      x,seed:=random(K,seedInitial)
      SEQ i=0 FOR N
        SEQ
          x,seed:=random(K,seed)
          out.number(x,2,out)
          out.string("*n",0,out)
      out.string("___",0,out)
      out.string("*n",0,out)
  :

  SEQ
    Genere(12)
    Genere(2)
```

```
    Genere(12) -- la souche est la meme que le premier cas et donne le meme tirage.
:
```

Compiling (`kroc -lcourse randomSample.occ`), and executing (`./randomSample`)produces the following trace:

```
 1
 3
 1
 8
 0
---
 8
 9
 6
 3
 4
---
 1
 3
 1
 8
 0
---
```

It is noticeable that the seed value allows to reproduce exactly the same random sequence of numbers. Thus, if we are to use this mechanism inside network simulators, we must vary the seed at each node. The unique `Identity` of nodes is a good way to do this.

Chapter 5

A NetGen-compatible map browser

This chapter will present two evolutions of the initial Netgen program for support of precise geographic positions and map browsing, then for displaying buildings or obstacles representations extracted from OpenStreetMap databases. The present tools allow to interact with the more common public information systems such as Google map and OpenStreetMap.

The map browser is a tool allowing to display various kind of maps and to represent locations of interest such as sensors set in the country. As this tool is developped on the same platform as NetGen, the procedures described in chapter 1 will apply to access the software:

- start a fresh image and ensure that the NetGen package is loaded with one of the last version (1.28.1.2.5 should work)
- open the store dialog from VisualWorks main window
- select GoogleMap package (we need to change this name)
- select version 1.15.5.2 or later, and type load from the pop-up menu

The initial window displays as shown figure 5.1.

5.1 Moving on the map

A predefined position is visible inside the `xtile` and `ytile` fields in the left column of the browser. Just below, a zoom factor is also provided. Whatever are these values, by pushing the refresh map button, the browser will download geographic information to be presented in the graphic pane.

All around this pane, four sidebars allow to move the graphics top, down, left, and right. The four rectangles at the corner of this view control moves on diagonal directions.

By changing the zoom factor, the absolute view size will decrease or increase. As an example going from 15 to 16 increase the level of details.

5.2 Loading networks

Networks are loaded from external files (further versions will allow direct selection from the interface).

Currently, GPX file format is used, as it is a very common way to describe set of points featured with attributes.

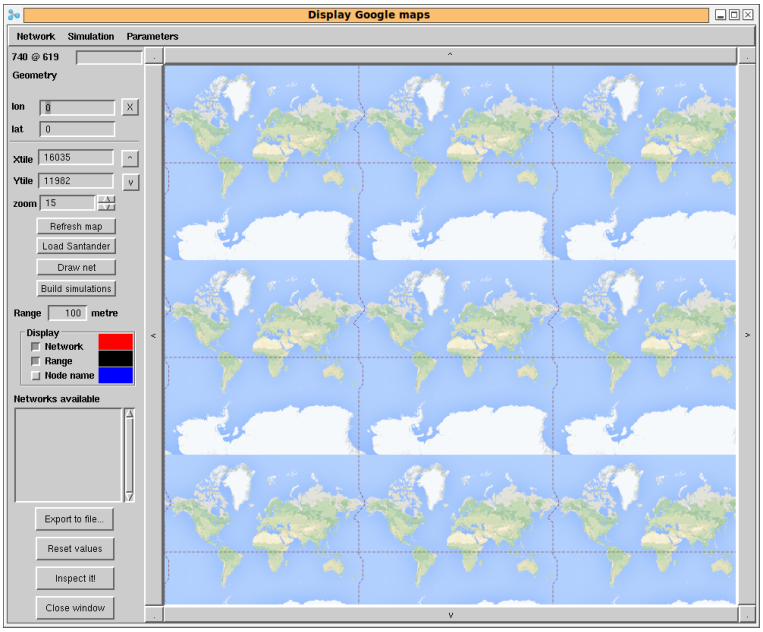

Figure 5.1: Initial view on the map browser: the right part displays tiles from the map from public servers. The left column displays geographic information and allows to control network presentation.

5.2.1 Scenario for loading informations

Suppose that by moving on the map, we have reached a particular region where a sensor network is setup or planned.

First, let us comments what is a GPX file. In this format, we find a header that keep information about the initial source of contents. As an example, a header from the Santander website could contain:

```
<?xml version="1.0" encoding="utf-8"?>
<gpx version="1.0" creator="NetGen for Santander">
  <metadata>
  <name>SmartSantander's sensors</name>
     <desc>Sensors in city: Santander, Spain</desc>
     <link>http://smartsantander.eu/</link>
     <time>2013-06-13T17:45:10</time>
</metadata>
```

After this there is a list of entries for each of the location documented by the file. In the case of our river, we will find tens of similar entries such as:

```
<wpt lat="36.679926936710501" lon="4.911090436802451">
  <name>H1-2</name>
  <sym>El Kseur</sym>
</wpt>
<wpt lat="36.682937769536323" lon="4.919011088180600">
  <name>I2</name>
  <sym>El Kseur</sym>
</wpt>
<wpt lat="36.686345105073165" lon="4.929854203839318">
  <name>J3</name>
  <sym>El Kseur</sym>
</wpt>
...
```

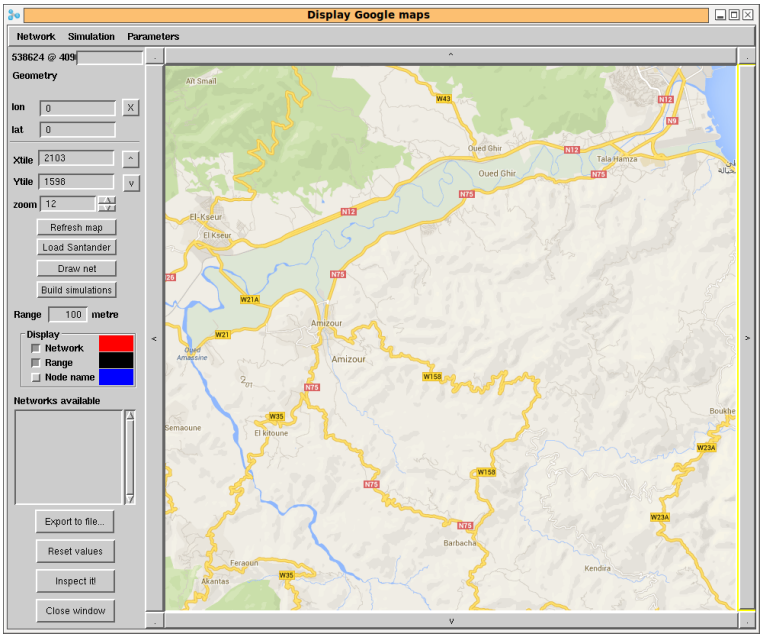

Figure 5.2: The browser is pointing to a North Africa river where sensors are to be installed for water observations.

This a very short information since no practical values appear from sensors. The file extraction presents three waypoints (from the initial purpose of the NMEA standard for GPS), with geographic coordinates as decimal expression of degrees. Following we find a name for this particular point, and a symbol to display.

5.2.2 Loading a network

By using the *Network* menu at top-left we can select the *Load Gpx* function that brings a dialog to select a GPX file. In our case, it is *soummam.gpx* file to remember the name of the river and the file format. The file is parsed and its contents appears as points on the graphic part.

The symbols appearing in the entries of the file are used to group waypoints together inside networks. This networks are shown in a list in the left column. They are selectable, and as an example, the network *El Kseur* is validated for display figure 5.3.

5.2.3 Network configuration

Several of the functions of NetGen as described in the previous chapter are avalaible from this front-end window which capabilities exceed the picking tool shown chapter 2. As an example, the range used to decide wether a sensor is connected to another can be defined using a dedicated numeric field. Furthermore, the present tool has precise knowledge about geographical points and related distances including display distances. Thus, the distance can be defined as meters conforming to radio capability specification.

On figure 5.3, the range has been tuned to the point where each sensor in the El Kseur network was reachable, giving a necessary range of 3 000 meters to include all the sensors. The window does not react directly to range modification, it is necessary to call *Draw net* button.

Some colouring functions ease the display of sensor names, range circles, and connectivities.

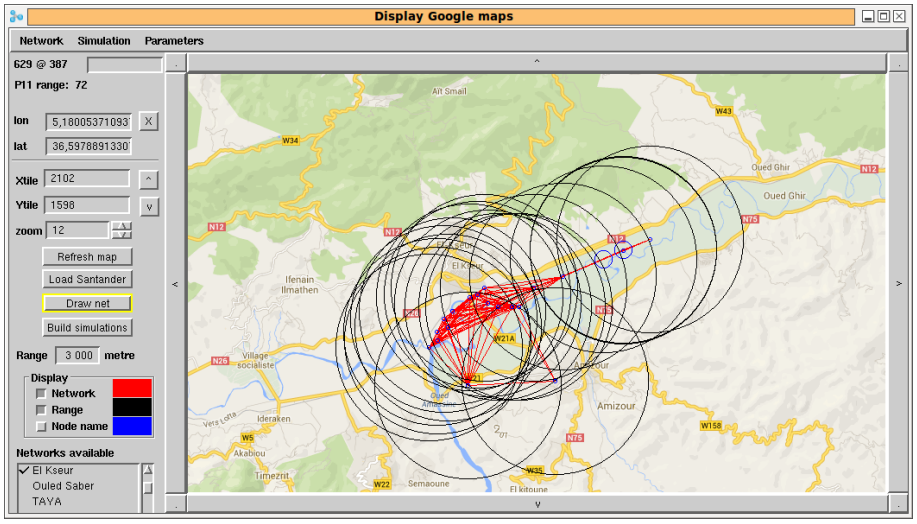

Figure 5.3: The browser is pointing to a North Africa river where sensors are to be installed for water observations.

The mouse location over a graphic presentation is tracked on the top-left of the window:

- point coordinates inside the window, changing to red when the mouse is precisely over a sensor.
- this case the logical name of the node is shown in the edit field.
- a second line presents the range and position of the closest sensor, or a communication channel in the case where the mouse is over such a channel.

5.2.4 Network generation

To reach Netgen functionalities related to the process graph specification, it is sufficient to depress the build simulations button: the specification is loaded in every Netgen window (chapter 2 and figure 5.4) for further use: building simulation, graph drawing (see figure 5.5), etc...

After this exploration we have learned that this network has 21 nodes with maximum connectivity of 16, with 232 communications channels.

Executing the simulation brings a value of 4 hops for the network diameter. We probably have interest to reduce the general range keeping the further nodes to a 3 000 m range. An important aspect behind sensor nodes and map is that the simulator generated in Occam or Cuda preserve the logic link between simulation processes and graphical representation. Thus a simulator can animate the graphical view from outside concurrent execution.

5.3 Loading building architectures

The package concerned with building representations is MapAccess one, also available from the same store server as for chapter 1.

In addition to this package, we need some files in the format of ”shapefile”. This time, we propose to do our scenario on the case of Brest city which administration decided to produce large description as 80 000 buildings set.

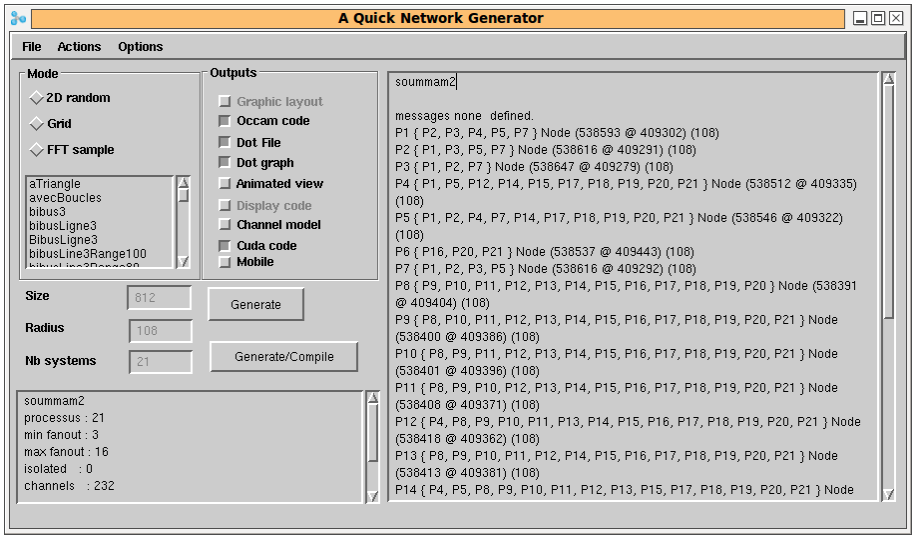

Figure 5.4: Netgen window with El Kseur specification and statistic.

- start a fresh image and ensure that the NetGen package is loaded with one of the last version (1.28.1.2.5 should work)
- open the store dialog from VisualWorks main window
- select MapAccess package (in the future, likely to be merged with the map browser)
- select version 1.6 or later, and type load from the pop-up menu
- open a system browser, look for the MapAccess package, and the comments for this package. It is a button in the middle of the browser window. This comment looks like the following sentences:
  Display maps from tile servers, like Google Map or OpenStreetMap. Georeference points on the map. Display objects from shapefiles. Library is located here: http://wsn.univ-brest.fr/MapAccess/library/libShapeFile.tar. Run 'make' to compile it. BMO shapefile is located here: http://wsn.univ-brest.fr/MapAccess/bmo/. Copy the two files shp and shx in the same directory.

  This comment is likely to change, but we will keep location allowing to download software for the package: dynamic libraries compiled for Linux and MacOSX, and shapefiles for Brest. We do not support Window currently.
- in your working directory, download the libraries:
  `wget http://wsn.univ-brest.fr/MapAccess/library/libShapeFile.tar`.
  Desarchive the tar file by doing tar xf libShapeFile.tar.
- and download the two shapefiles:
  `wget http://wsn.univ-brest.fr/MapAccess/bmo/bati-WGS84.shp`
  and
  `wget http://wsn.univ-brest.fr/MapAccess/bmo/bati-WGS84.shx`.
  (ftp, curl, or any web browser can do similar work)

5.3.1 Checking the configuration

To check the installation, it is best to open a system browser on class `ShapefileReader` (figure 5.6). This class uses directly the public domain library for assessing file conforming

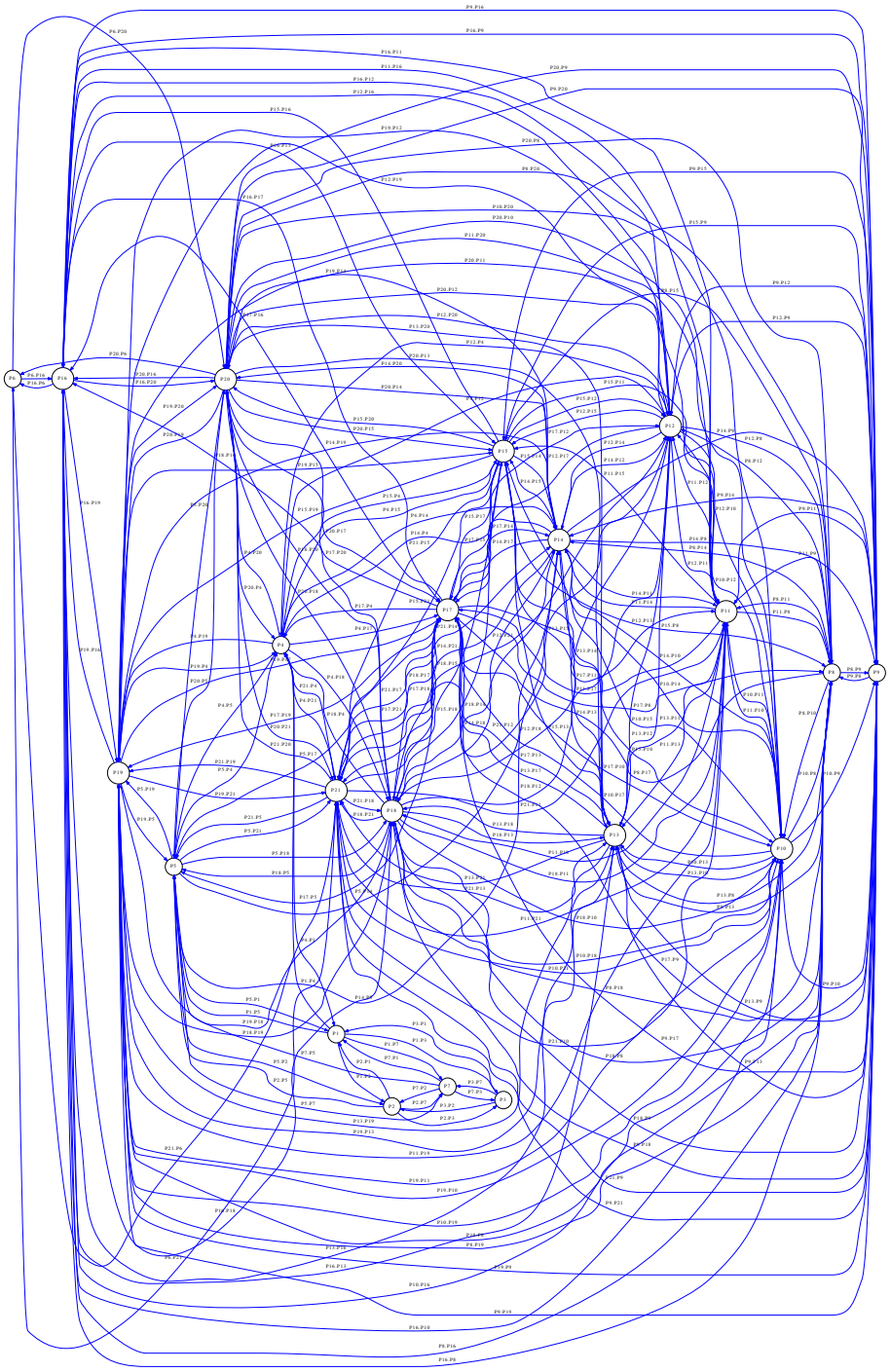

Figure 5.5: Logic graph for El Kseur location network.

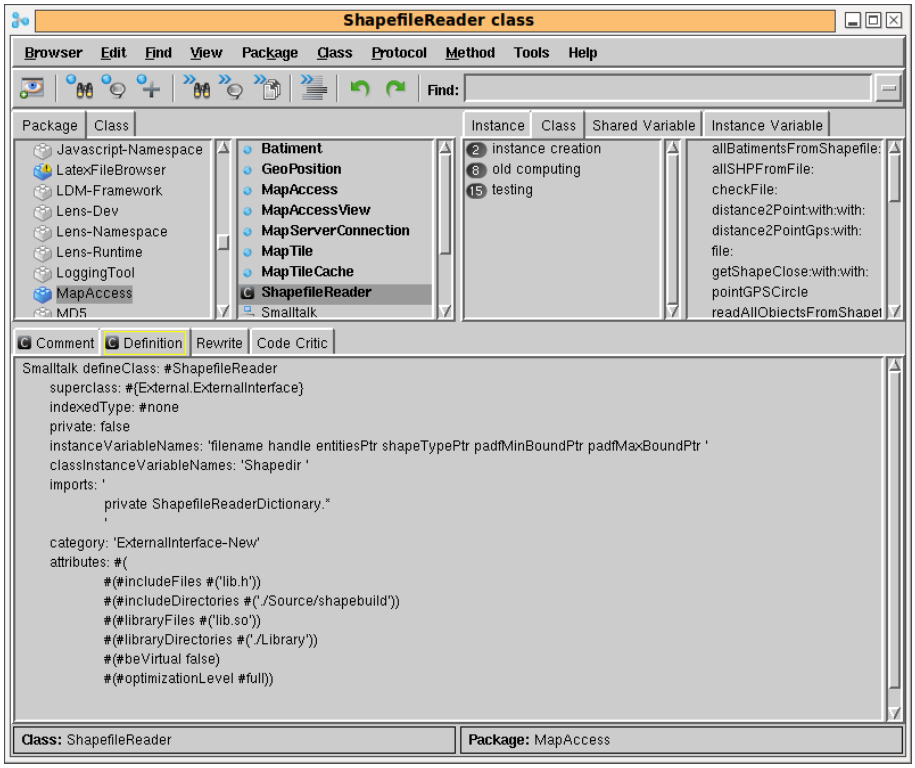

Figure 5.6: Path configuration for the dynamic libraries

to the shapefile format. It translates external definitions into objects to be displayed on a derivative of the map browser interface.

It can be necessary to adapt the two directories for includeDirectories, for libraryDirectories to the platform, observing that a regeneration can be possible based on this public software.

5.3.2 Interface opening

By selecting the Tools menu from the main window, then choosing ”Universite... MapAccess” the new interface opens. This interface is quite similar to the map navigation interface, at the exception at the left column. Having the interface open, by clicking Reset values, then Refresh map, we obtain a default view on Brest city (figure 5.7).

5.3.3 Loading shapefile

The button Open shapefile brings a dialog where a .shx file must be selected The figure 5.8 shows Brest city map with more than 80 000 buildings represented as polyline 2D objects.

5.3.4 Browsing the city

The capabilities of the map browser exists in this preliminary tool. As exemple, one can zoom and move the display view changing the level of details. The geographic coordinates being preserved, it is possible to focus very precise situations to enable simulations and computations. See figures 5.9 and 5.10.

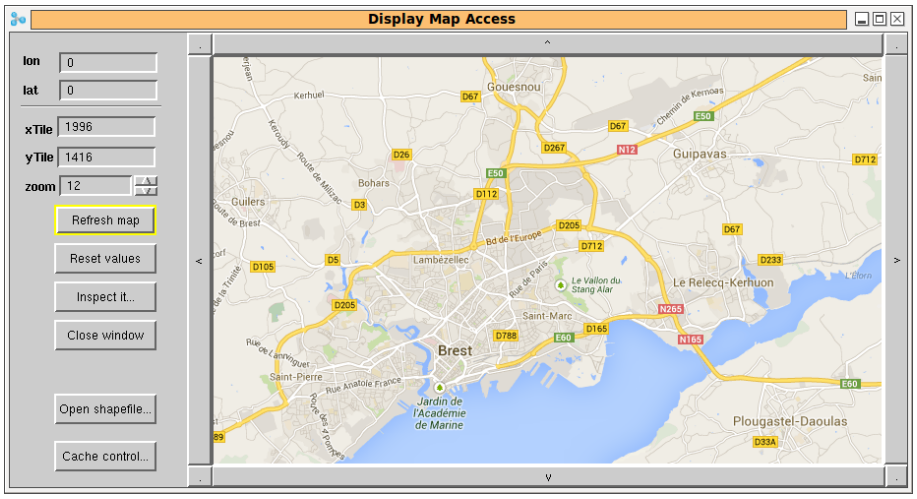

Figure 5.7: Brest presentation before loading a shapefile

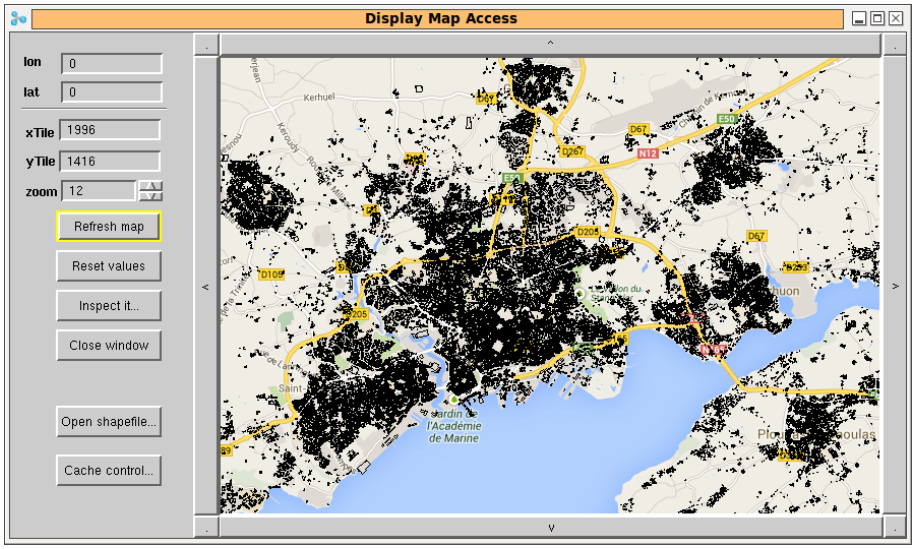

Figure 5.8: Brest presentation once the shapefile is loaded

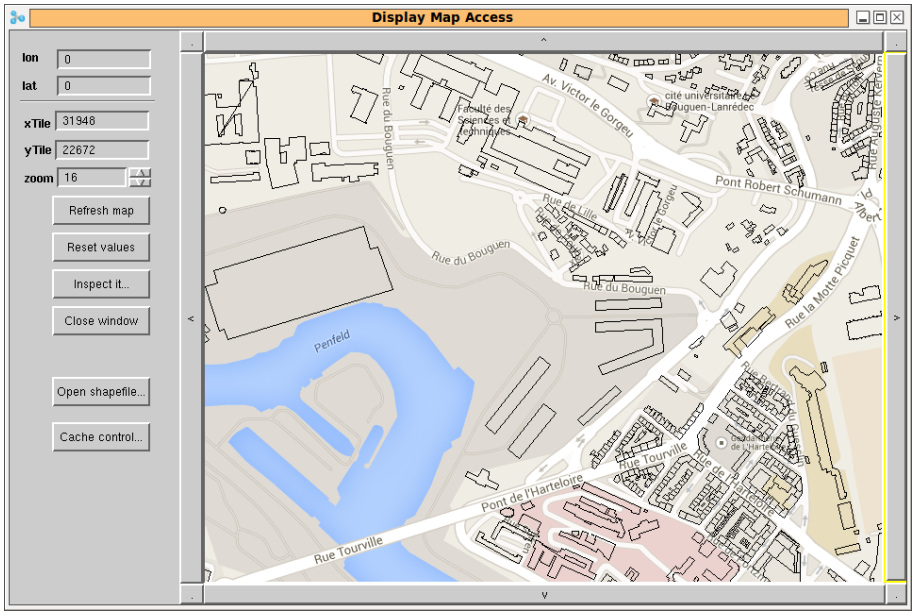

Figure 5.9: Brest university and river Penfeld after loading BMO shapefile, Google Maps

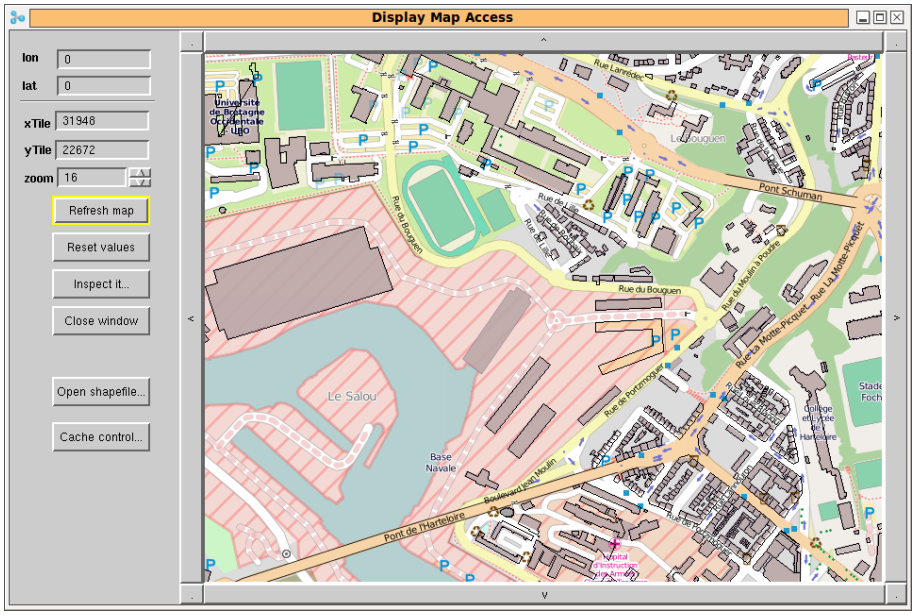

Figure 5.10: The same view as for figure 5.9, using OpenStreetMap.

Switching between map layout is possible using a dialog installed under the cache control button. This dialog also allows to empty a cache used in the present software to speed up access to distant map tiles.

5.4 Algorithms

5.4.1 Layers

The information displayed in this application is divided in two layers : the base map, provided by a map server, and the network, locally computed. More generally, displaying is composing an image from several information sources. The composition is understood as a projection of differents layers. In that sense, the building computation explained in section 5.3 is just a particular case of a general process. Network presentation and activities, mobiles moving on their path are other cases.

5.4.2 Tiles of the base map

The map is divided into tiles, which are the same size (a square of side 256 pixels in our application). When we put together adjacent tiles, we can build the map of an entire zone. The system of tiles can be represented as a pyramid, according to the zoom level. At zoom zero, there is only one tile, at zoom one, there are 4 tiles, at zoom 2, 16 tiles. The computation of the number N of tiles given a zoom level Z is as follow:

$$N = 2^Z.$$

The tiles are retrieved from a tile server, like Google map or OpenStreetMap. They are downloaded by http client, the url gives the position of the tile over the map, and the zoom level.

Tiles are portion of map, with a zoom level which gives more or less details on the area covered.

5.4.3 The projection question

The Earth is spherical, whereas maps are printed on a plane surface. The method which is used to transform spherical coordinates into orthogonal coordinates is called projection.

One question to be considered in the first rank is how a 3D topology can be represented on a 2D surface. This is called the projection system, and one being used on most of the Map Access software is Spherical Mercator. For this application, this projection transforms geodetic coordinates (longitude and latitude) in meters. Thus, the global map is projected into a square, which is divided into tiles. Tiles have the same size, usually 256×256 pixels. The zoom level defines the number of tiles produced. At zoom 0, there is only one tile, at zoom 1 there are 4 tiles, etc... There is a formula to convert spherical coordinates (longitude and latitude, expressed in degree) into orthogonal coordinates (x and y, expressed in metres). The formula to convert from one sytem to the other is as follow :

```
latLonToMeter: lat lon: lon

        | mx my |
        mx := lon * self originShift / 180.0.
        my := ((90 + lat) * Double pi / 360.0) tan ln / (Double pi / 180.0).
        my := my * self originShift / 180.0.
        ^mx @ my
```

Then, we have to find the tile corresponding of the (x, y) coordinates. First, we compute pixel coordinates on the map, from the (x, y) coordinates. The formula in Smalltalk langage is given below.

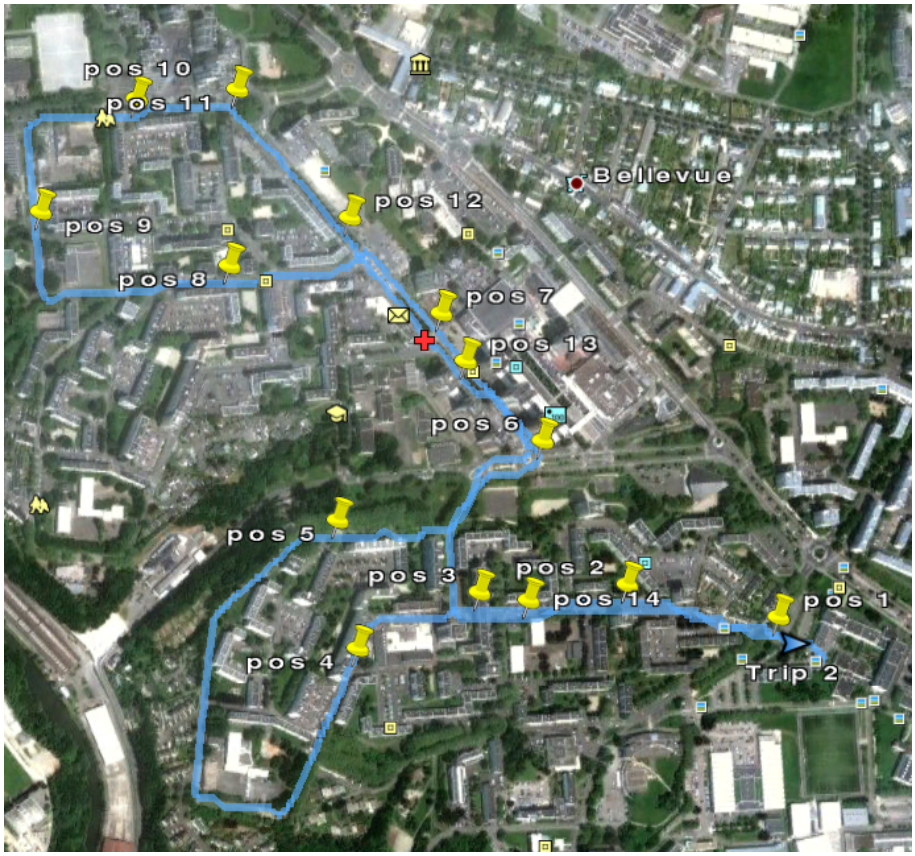

Figure 5.11: A trip in Brest city, along a bus line, with a GPS data logger : the path of the car where the tool was installed is printed in blue, the bus stops are indicated by the yellow pins. The GPS is accurate enough to discern the both side of the street.

```
meterToPixel: xy zoom: zoom

        | mx my res px py |
        res := self resolution: zoom.
        mx := xy x.
        my := xy y.
        px := (mx + self originShift) / res.
        py := (my + self originShift) / res.
        ^px @ py
```

5.4.4 Tile Assembly

In the display surface

5.4.5 Accuracy

The GPS device has a maximal accuracy of 5 m, thus the coordinates are given with 7 decimals. A small error in coordinates can lead to several metre shifts on the map. In order to achieve correct calculus with this, we used fixed-point computation in Smalltalk. Thus, the results are not rounded, which increase accuracy. In figure 5.11, we can discern the both sides of the street, those data was provided by a 5-m accuracy GPS, which is the maximum reachable for civilian use.

Chapter 6

QuickMap : another NetGen-compatible map browser

Note : QuickMap is developed by Pierre-Yves Lucas as a geographic frontend to NetGen.

This chapter will present QuickMap, an evolution of the tool presented in chapter 5 designed to support precise geographic positions and easy map browsing.

A movie is available at `http://wsn.univ-brest.fr/QuickMap/` that demonstrates the tool in action to describe and control field of sensors, working under the control of satellite.

QuickMap has also support for displaying buildings or obstacles representations extracted from OpenStreetMap databases. The present tools allow to interact with the more common tiles servers. An example of such a server is OpenStreetMap reachable at http://tile.openstreetmap.org[1].

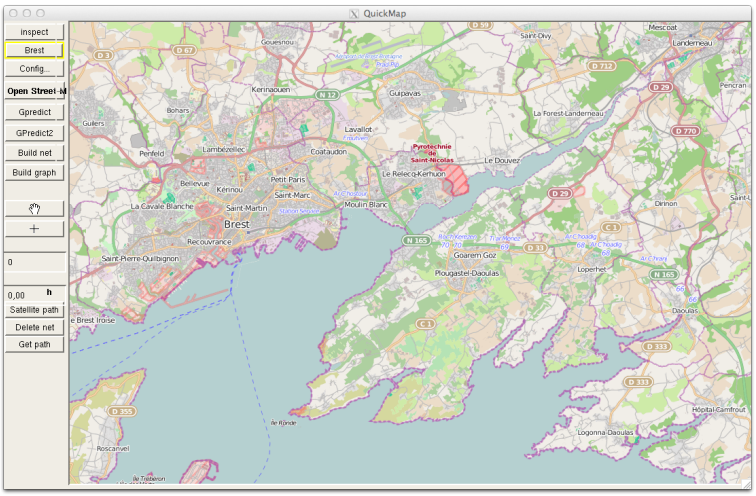

Figure 6.1: Initial view on QuickMap version for satelite control, pointng to default Brest location. The user interface allows to move directly maps using the hand cursor. The sensor wheel controls depth in the map system, here OpenStreetMap.

The map browser is a tool allowing to display various kind of maps and to represent locations of interest such as sensors set in the country. As this tool is developped on the

[1] See the explanations about access agreement, keeping in mind that the present software is a reserch investigation, and not a production application

same platform as NetGen, the procedures described in chapter 1 will apply to access the software:

- start a fresh image and ensure that the NetGen package is loaded with one of the last version,
- open the store dialog from VisualWorks main window menu ba,
- select QuickMap package ,
- select last version, and type load from the pop-up menu

Once the package is loaded, launching is achieved from the Tools menu, with a number of variants available (figure 6.2).

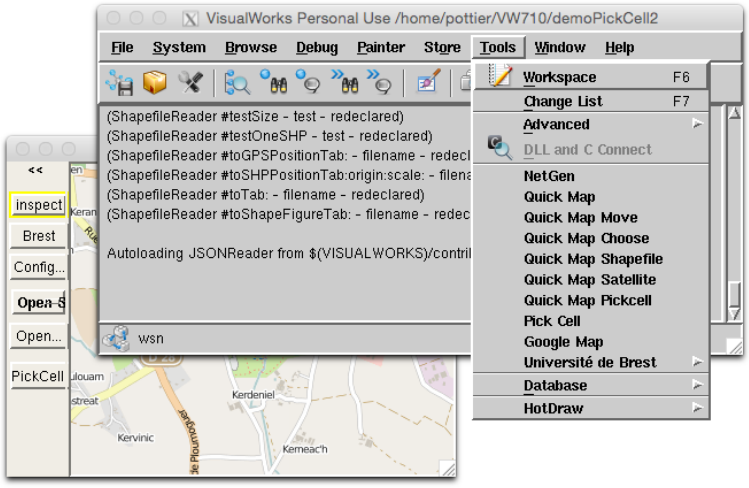

Figure 6.2: Tools menu after loading QuickMap, select one of the interfaces from here.

6.1 Short presentation

Figure 6.1 shows a view on a QuickMap windowi with a number of control button on the left, and map display on the right.

The movie proposed at `http://wsn.univ-brest.fr/QuickMap/` is a sequence representing the current tool status (figure 6.3):

Moving the map : shows horizontal and vertical moves on an OpenStreetMap map system.

Launching GPredict : external software that allows to select satellite and pipe informations to QuickMap.

Following Satellite : these values are collected and presented on QuickMap, the red ball figures the selected satellite.

Specification of a sensor field : some sensors are positioned in front of the satellite path.

Building networks : build button ask computation of edges representing communication links establishment and disconnection.

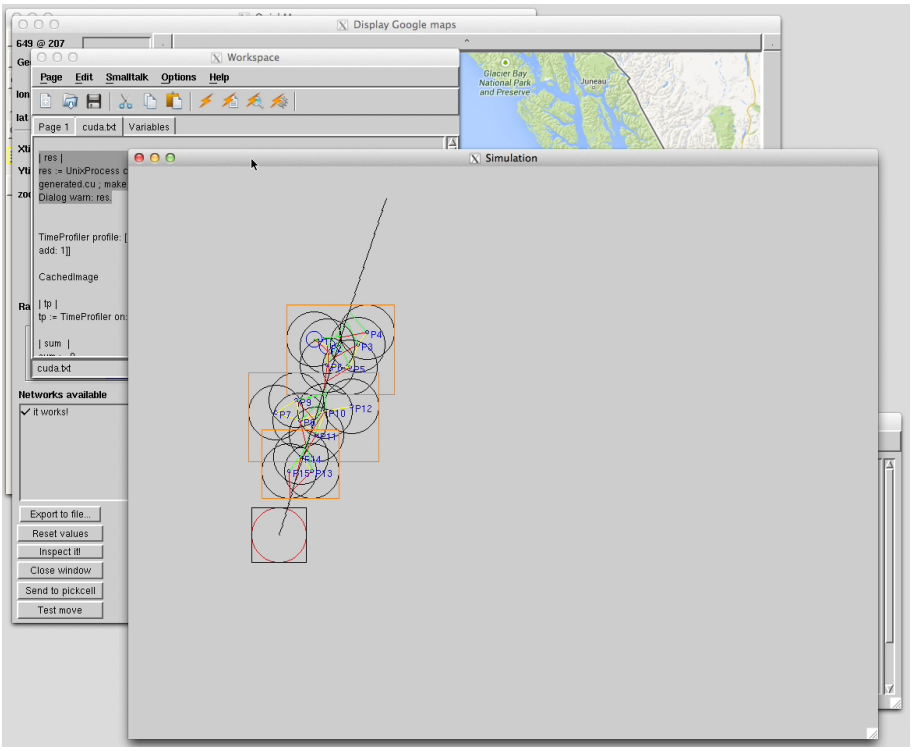

Figure 6.3: End of simulation, showing the satellite path and subsystem bounding boxes.

Network synthesis : using the old version of the browser, the communication ranges can be modified, then the problem is passed to NetGen for simulatioon (CUDA).

System simulation : satellite and sensors work together, synchronously. The distributed program start at the same step on the ground and in the air. Bounding box for controlled sensor systems are computed and displayed.

6.2 Building sensor fields

Movie available at `http://wsn.univ-brest.fr/QuickMap/`, select some version of `fieldOfSensorsSpec.mov`. The flow is as follows:

Launch QuickMap, option satellite : then navigate to a zone where the field must be installed,

Select a zone : select the crosshair cursor, type the number of sensors in the input field on the left, and clic top left, then bottom right.

Following Satellite : these values are collected and presented on QuickMap, the red ball figures the selected satellite.

Build graph : displays a net with a default range.

Build net : open a GMap window, from which you can :

1. refresh map to your sensor field
2. change the radio range

3. draw resulting net (see Figure 6.4)

Generate abstract specification : open a NetGen window from the general Tool menu, then press build simulations from GMap

Save simulators : change the network name to something like ”enezEussa”, or ”Sahara”, select generation options, and accept-and-save from the pop-up menu.

Observe results : files are in the local Generated directory, Occam, Cuda, or graphviz formats.

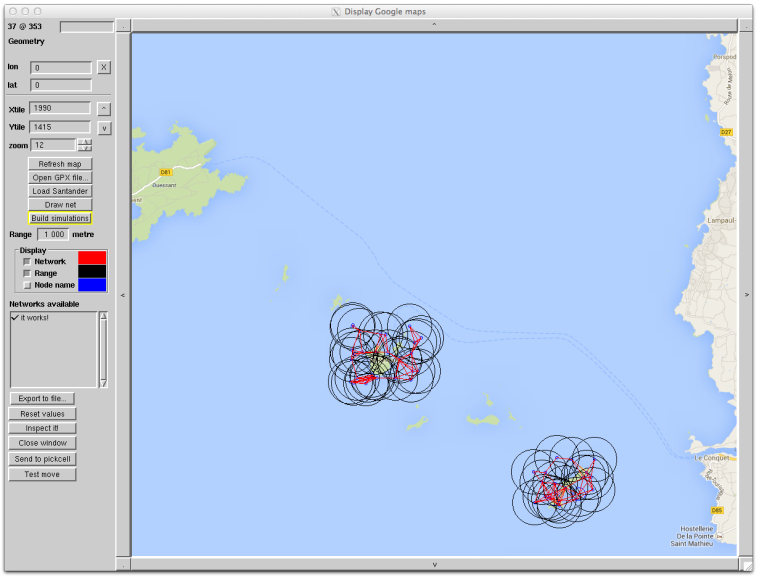

Figure 6.4: Random sensor field over Iroise islands, 1Km radio range, two fields of different sizes. 50 random positions. Simply press + as many time as necessary to obtain several rectangle, and thus, several fields.

Chapter 7

Binding simulation to GUIs (back annotation)

Combining physical simulation and sensor network simulation allows to verify the accurracy of the sensing process in relation with the physical process. In most of the cases, the two activities are independent, but there are known situations where a control loop exist.

This chapter will shortly discuss preliminary works where geographic data are extracted (see chapter 5), analyzed, and processed to simulate the the physical process:

1. loop between graphical planning, simulation synthesis, and graphic interfface control from simulation,
2. case of a mobile moving inside a set of sensors,
3. case of cellular automata representing physical process.

In these situations the physical process spread over 2D or 3D spaces, that are divided into a number of adjacent cells. One solution to keep track of evolutions is to use massive parallelism with a good computation candidate being cellular automata.

7.1 Binding simulator to the map browser

Figure 7.1 shows a view on a map interface connected to an Occam simulation. This is a view of the Santander network retrieved in real time from `http://smartsantander.eu`, with the sound network selected for display. Communication links are initially shown as red line, but to demonstrate the simpulation effect, we switch to the green color for each node sending a message.

It is noticeable that the small networlk at the bottom left is finished, while le big network is still revealing information.

This section will explain how the simulation, and even real messages from real sensors can interact with the graphic view. Figure 7.2 presents the system organization, with messages multiplexed to an Occam relay for an external process running Smalltalk. In the Smalltalk image, messages are decoded and actions are taken to display visually changes from simulation.

7.1.1 Calling back the GUI

The architecture description code source need support from an Occam mechanism allowing to fork external processes. The setup below will do, coupling a byte channel to the i/o stream of this process.

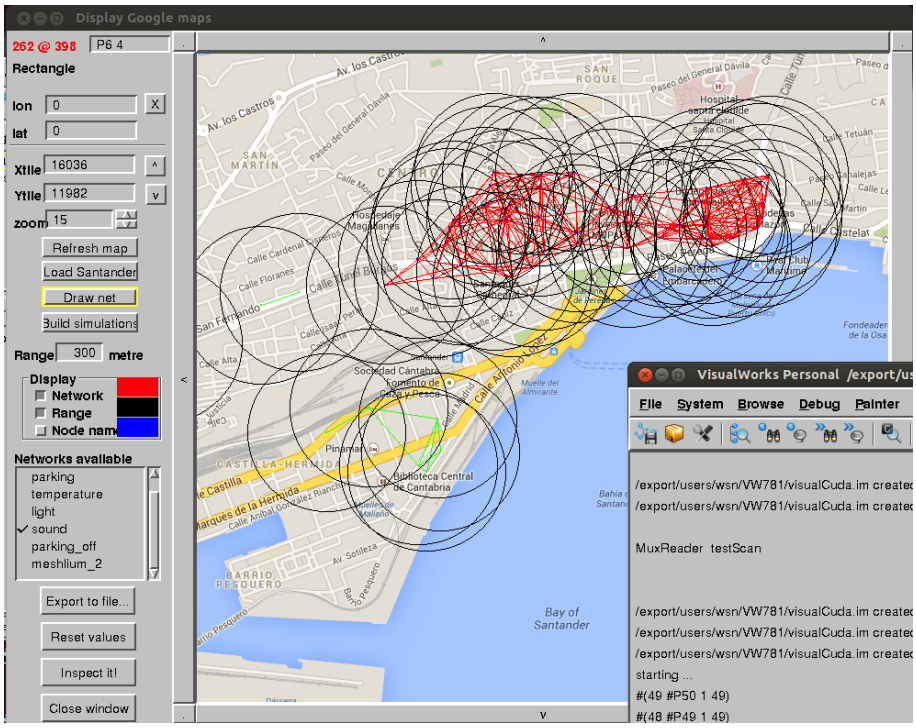

Figure 7.1: Feed back from concurrent simulation to Map Browser interface

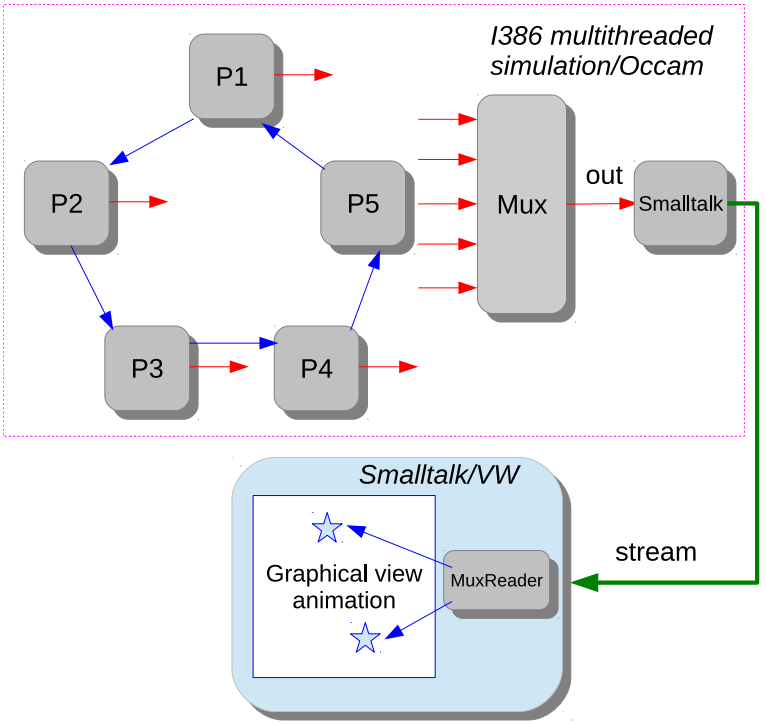

Figure 7.2: System organization between simulator and graphical interface

```
PROC Smalltalk(   CHAN OF BYTE in , out , err )

  VAL [2][] BYTE Prog IS [ "/usr/local/vw7.8.1nc/bin/linux86/visual", "./visualnc.im" ]:
  VAL []BYTE arg IS  "./visualCuda.im":
  [2][100] BYTE Prog2 :
  [1]ENVIRONMENT envArray:
  INT result:
  SEQ
    -- nettoyer le tableau
    SEQ i=0 FOR 100
      SEQ
        Prog2[1][i] := (BYTE 0)
        Prog2[0][i] := (BYTE 0)
    -- copier la chaine de commande
    SEQ i=0 FOR SIZE Prog[0]
      Prog2[0][i] := Prog[0][i]
    -- copier la chaine argument
    SEQ i=0 FOR SIZE arg
      Prog2[1][i] := arg[i]
    -- configurer l'environnement
    proc.setenv (envArray[0], "VISUALWORKS" , "/usr/local/vw7.8.1nc")
    -- demarrer smalltalk
    out.string(Prog2[0],0,out)
    out ! ' '
    out.string(Prog2[1],0,out)
    out ! '*n'
    proc.wrapper(envArray, Prog2,  in , out , err, result)
:
```

Inside the parallel construct, we now need to start a process that will fork a Visualworks image outside. The MuxToST channel will send information from the Occam simulator multiplexer to the external Smalltalk GUI.

```
[MaxNodes]CHAN OF BYTE toMux:
 CHAN OF BYTE MuxToST:
 PAR
   Smalltalk(MuxToST,stdout,stderr)
   Mux(toMux,MuxToST)
   Node(P1.in, P1.out,0, toMux [0])
   Node(P2.in, P2.out,1, toMux [1])
   Node(P3.in, P3.out,2, toMux [2])
....
```

7.1.2 Passing contextual information to the simulator

.

For each process generated from NetGen, there is an entry in a varaible array. The code below shows part of this array, where the data is simply the name of each process, as it appears in the network specification.

This name is a minimum information to advertise a GUI or tracer about the identity of an emitting process.

```
VAL [51][3]BYTE NetProcess IS [   "P1 ", -- id: 1
  "P2 ", -- id: 2
  "P3 ", -- id: 3
  "P4 ", -- id: 4
  "P5 ", -- id: 5
  "P6 ", -- id: 6
  "P7 ", -- id: 7
  "P8 ", -- id: 8
  "P9 ", -- id: 9
  "P10", -- id: 10
....
]:
```

The Mux relays information by sending the index of the channel, then copying the entire line to the output.

```
PROC Mux([]CHAN OF BYTE muxTab, CHAN OF BYTE out)
  BYTE c:
  INT t:
  INT t64:
  SEQ i=0 FOR ( MaxNodes)
    ALT i=0 FOR SIZE muxTab
      muxTab[i] ? c
        SEQ
          out.number(i,4,out)
          out !'*t'
          out ! c
          WHILE c <> '*n'
            SEQ
              muxTab[i] ? c
              out ! c
:
```

7.1.3 Demuxing in Smalltalk

Lot of things can occur there by using Obectt oriented facilities for sensor attributes. In this demonstrator, we just decode lines sent from Smalltalk, select the sensor, displays its name, change colour all around, an place the mouse over its location.

A class MuxReader has been developped for this, and the system code testScan for decodint on the stream is showne below:

```
testScan

  | connect line info ugm i delay finished |
  self allInstances do:
      [:mr |
      mr streamIn close.
      mr streamOut close].
  connect := self connect.
  ugm := UIGoogleMap new.
  ugm open.
  (Delay forSeconds: 60) wait.
  Transcript
    cr;
    show: 'starting ...';
    cr.
  i := 0.
  delay := 0.
  finished := false.
  [6 * 60 timesRepeat:
      [(Delay forSeconds: 1) wait.
      delay := delay + 1]] fork.
  connect
    process:
      [[finished or: [connect streamIn atEnd]] whileFalse:
          [line := connect streamIn upTo: Character cr.
          info := connect scanLine: line.
          finished := info size = 1.
          finished
            ifFalse:
              [i := i + 1.
              Transcript
                show: info printString;
                cr.
              [ugm changeMousePositionFromName: (info at: 2)] value.
              Processor yield.
              (Delay forMilliseconds: 1000) wait]].
      connect streamIn close.
      connect streamOut close]
```

```
        fork.
^connect! !
```

Chapter 8

PickCell: from image analysis to physical simulation

The associated software is available in the Pickcell package pn Store at http://wsn.univ-brest.fr. Use is similar to chapter 6 explanations:

- *load pickcell, then load a file image from the application window seleted from Tool menu,*
- *load quickmap, then select the quickmap pickcell variant from tool menu, to enable pickcell from an OpenStreetMap view.*

Creation of sensing machines working in the environment necessitate a validation in the context of physical scenarios. Some of these scenario are flooding, insect clouds, pollutions, fires.

The focus of this chapter are tools and methods allowing to produce inputs and organizations for physical simulations. These simulation involve lot of computations. A choice is to use space and time discrete models such as cellular automata. It is also expected that physical simulation will cooperate with network simulations in various ways: production of stimuli on sensors, or modificaton of the sensor network itself. Figure 8.1 displays the general scenario with the left branch being discussed here.

The interest of image anlysis is to produce sets of regions that share similar characteristics. Analysis follows a conventional flow, starting from a picture coming from photographies, maps, radar images, to obtain regions of interests, on which physical simulation will take place.

A reference for processing on such regions are cellular automata reproducing fire expansions of phenomenon consuming vegetation. Figure 8.2 is extracted from a movie demonstration where ”fires” are started randomly to ”eat” such vegetation report [1]. This preliminary work was achieved on an Nvidia GPU.

The chapter will provide details on the production of systems representing the physical process working in harmony with the sensing systems.

8.1 Image processing flow for cellular system synthesis

Thus, modeling physical regions automatically, or semi-automatically, following physical process characteristics, is certainly critical to lead both physical and control network simulations jointly, and synchronously. One can think of this as a sampling technique of the physical process sharing a clock with the sensing network. Cellular automata are one way to implement the real world simulation, starting from initial states and regions.

A general flow to obtain such regions is shown figure 8.3 :

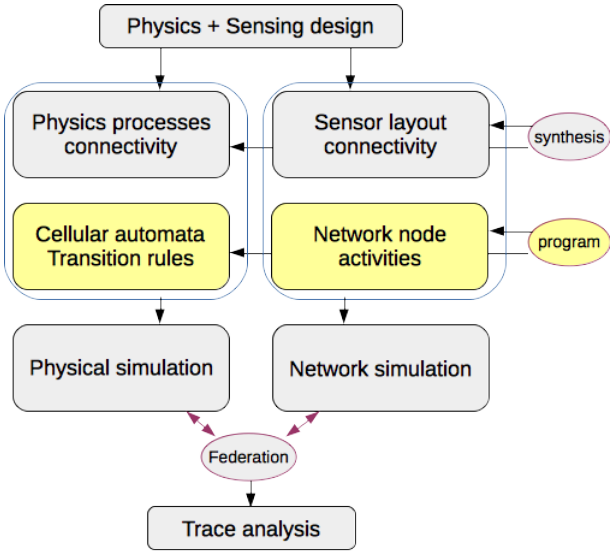

Figure 8.1: General flow for simulation with the physical simulation and the network simulation. Both share coordination informations such as geographical or geometrical points, clocking system, and they can be coordinated during the simulation.

Figure 8.2: Example of a fire simulation using Cellular Automata cells that have states such as: vegetation, burning, ashes.

1. preprocessing images
2. segmenting images into blocks
3. recognition of similar cells and grouping into regions
4. processing regions an obtaining skeleton images

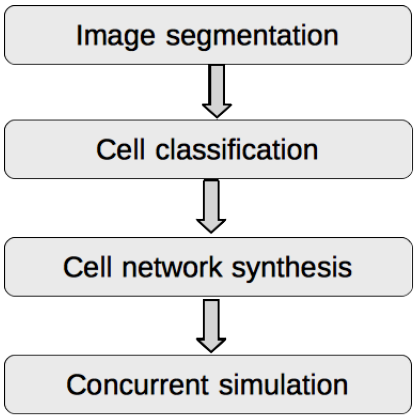

Figure 8.3: Preparing cellular automata from image analysis: synthesis flow.

Following this flow, higher level objects can be obtained, still with geographical definition in the case of maps, or satellite imagery.

8.1.1 Preprocessing for image preparation

In the case of maps, geographic information is yet presented in a comprehensive way. However analyzing maps is still useful to obtain information without the direct contact of an initial information system (GIS). More difficult is the case of satellite or air images, because the synthesis of objects necessitate pre processing. Figure 8.4 shows an example of procesing achieved using common tools for management of photographies. The initial view is a satellite image (left), while the right part displays an improved view, with better contrasts.

These representations are coming from a very common image processing software allowing to change contrasts and colour mapping to fit tne necessity of the recognition. Image processing paramers show the Red Green Blue statistic values in the original and the modified image, with a better use of the value range in the second case (Figure 8.5).

8.1.2 Segmenting the image into cells

To obtain regions, it is necessary to group zones of the image based on similarities of different kind. The first operation is a fragmentation into blocks of different size and geometry. Of course, squares, or rectangles are the simplest way to proceed, but other fragmentation techniques could also be suitable and ease further stages in simulation. Cellular automata propose as example an hexagonal shape where each cell has 6 direct neighbours.

PICKCELL has been implemented to ease this fragmentation, in relation with further networks synthesis operations.

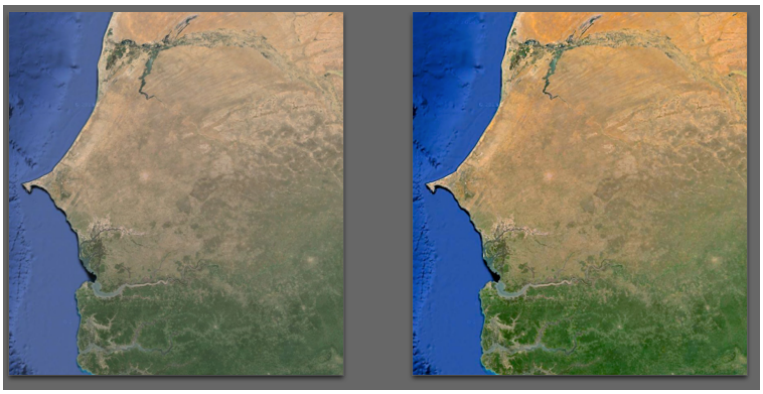

Figure 8.4: Modifying original image for better contrasts or coulour selections. Image is a satellite view from Google Maps.

Figure 8.5: RGB colour statistics for the left original picture, and its adaptation (right). The software is Apple iPhoto, standard to any MacOs computer. Similar functions could be obtained from Linux GIMP, and probably reimplemented from a Palette tool in NetGen.

This tool reuse the *Picking* framework presented Figure 2.5. Beside the capability to load images, and specify sensor positions, there is the ability to install a rectangular grid over the image.

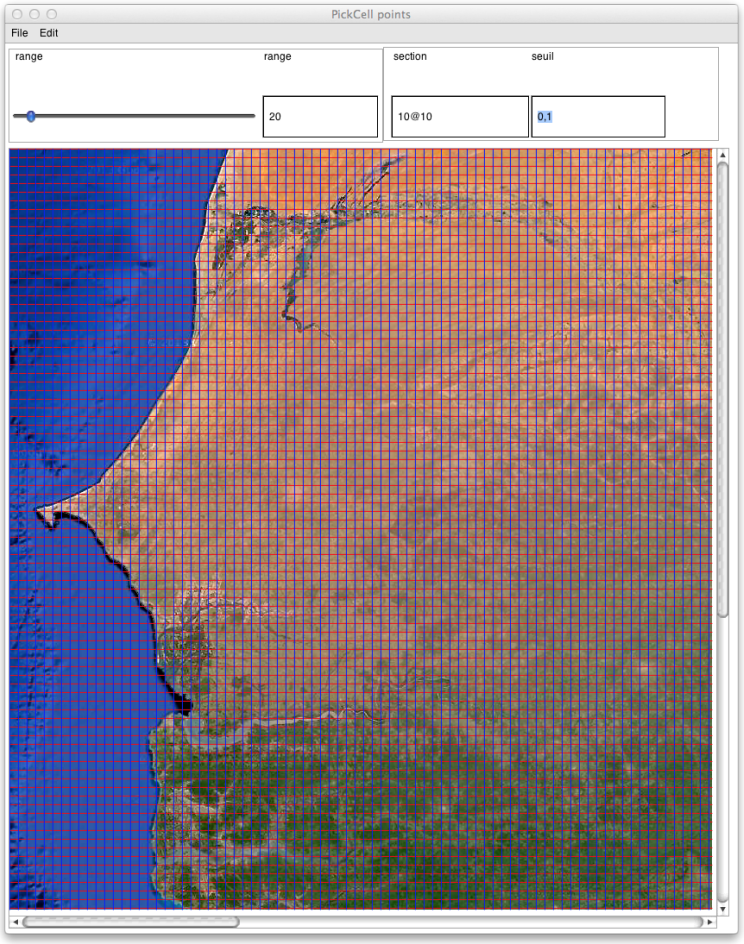

Figure 8.6: Using a 10@10 grid, the image is split into 10×10 pixels. Following the application, it is possible to specify $x \times y$ grids with $x \neq y$.

Having split the image into cells, it is now possible to classify thems around common properties.

8.1.3 Grouping cells into classes

Computing statistics

From the menu shown Figure 8.7, we obtain a new tool for region specification and manipulation. As for the grid in segmentation, a new parameter allows to divide the cell space according to pixel distributions.The choice of group distribution can be more or less sophisticated. To start by the begining, the following algorithm is applied to the whole cell grid, to obtain a set of $min, max, mean$ parameters on each colour :

```
STEP 1
        for each cell in the grid
                for each pixel in the cell
                        for each colour component in { R, G, B}
```

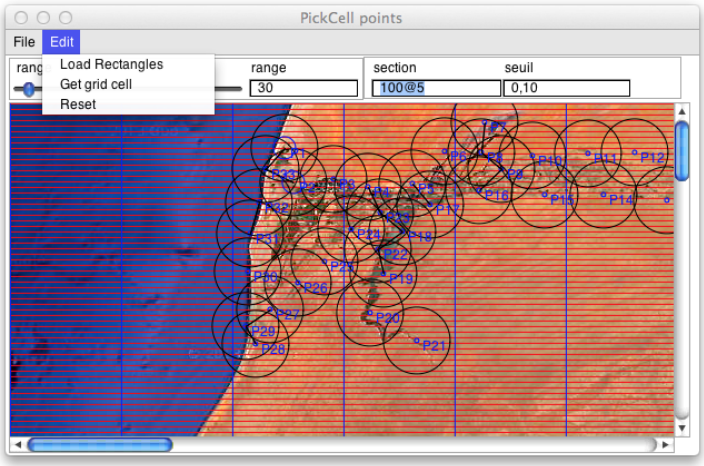

Figure 8.7: Rectangular grid, and sensor geometry specification. The Edit Menu *Get Grid Cell* allows to go a step forward in the flow to group cell togethers and form *regions* of interest. Section 8.1.3

```
                                update (min(colour))
                                update (max(colour))
                                update (sum(colour))
                for each colour component in { R, G, B}
                        update (mean(colour))
```

Following this step, we have 3×3 parameters set in each cell, thus allowing to compute global image characteristics that will reflect the efficiency of the preprocessing:

```
STEP 2
        for each cell in the grid
                for each colour component in { R, G, B}
                        update(minGlobal)
                        update(maxGlobal)
                        update(minMeanGlobal)
                        update(maxMeanGlobal)
```

After this step and for each component in RGB space, we have global $[min, max]$ measures for:

- the value taken over the image (see Figure 8.8)
- the mean of values in each cell

Grouping cell together

We can use the values computed in each cell and globally to group cell togethers. For the minimum and the maximum, and even the mean in each cell, a simple algorithm is as follows:

```
STEP3
        decide on a partition in N>1
        for each component in RGB
                discard any value out side the  [min,max]   interval found for the component
                produce N adjacent sub intervals   [min,max]
```

In the case of $N = 2$ the RGB min, max, or mean values in each cell will allow to *classify* each cell in an unique way in a 2^3 interval space $(R_0, R_1) \times (G_0, G_1) \times (B_0, B_1)$.

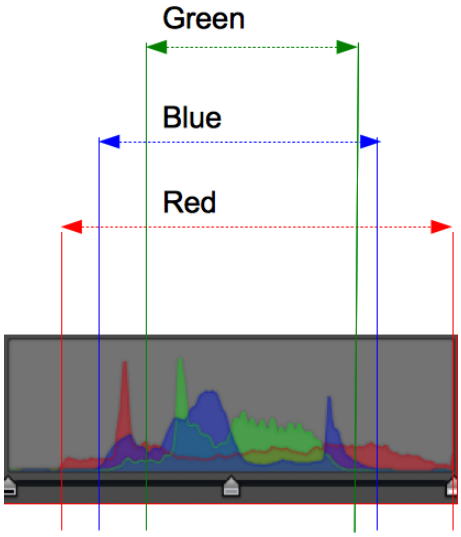

Figure 8.8: From the preprocessing results shown Figure **??**fig:colours1+2, we can guess min(colour), max(colour) for each component in Red Green Blue.

The *cube* of coordinates go from $0,0,0 = 0$ to $1,1,1 = 7$. By selecting *Get grid cell* shown Figure 8.7 the classification tool appears.

What does this too is to allow the selection of N, then each cell is tne image is assigned to an interval depending on its values. By selecting codes as defined above, the corresponding regions appear an part of the original image.

```
STEP4
        for each interval allocate a collection to record cells pertaining to this interval
        for each cell in the grid record the cell in its interval with the geographic position
```

8.2 Cell system synthesis

8.2.1 Weaving a network

Given a cell system produced from a class, grouping several classes, or from a whole image, we can now consider to produce a process organization. Taking one node, there are many way to decide which neighbours are visible, and this task can appear simple if we use a connexion pattern. Cellular automata use these patterns to define neighborhood.

Examples of neighborhood are Von Neumann cross, or Moore square. Mathematical morphology methods allow these neighborhood to be arbitrary shapes. As an example two line segments could figure a plane in the sky.

To build a cell system, the class is swept, row by row, column by column. For each existing cell, a process is created.

In a second stage, the class is again swept cell by cell. For each cell C, the neighborhood positions are searched inside the class. For each existing neighbour, links are established with the center C. This way a network of celles is weaved according to the neighborhood and the class geometric organization. Figure 8.11 displays the Cell class browser with choice of standard neighborhhod at the bottom.

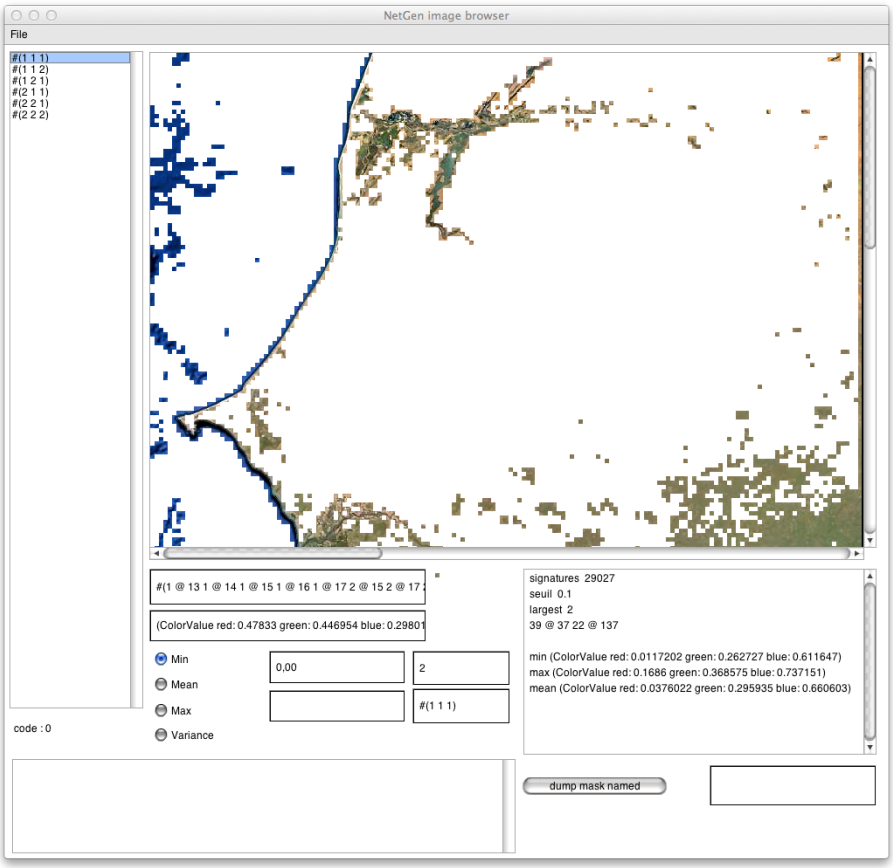

Figure 8.9: Pixel class browser displaying the region corresponding to code 0 : lowes interval in each component of R, G, B. Shore and rivers appear as dark cell when the minimum criteria are selected. The partition is N=2, code is 0, for (R=1,G=1,B=1).

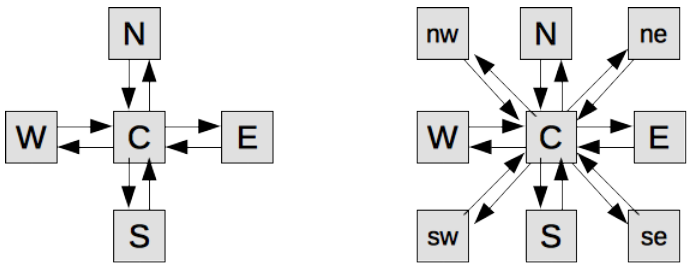

Figure 8.10: Von Neumann and Moore neigborhood with distance = 1

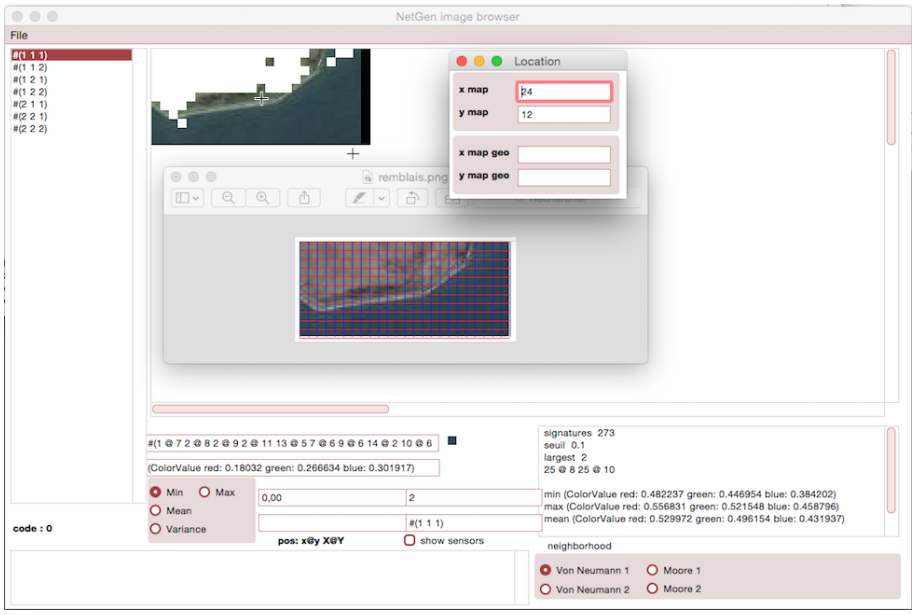

Figure 8.11: Segmented image and its distribution into classes, code 0 selected. Picture shows a cross cursor above one of the cell ant the location of the cell. This cell has neighbours at North, South, East and West, and thus has a full Von Neumann neighborhood. However a Moore neighborhood would be incomplete due to cells lacking in the north

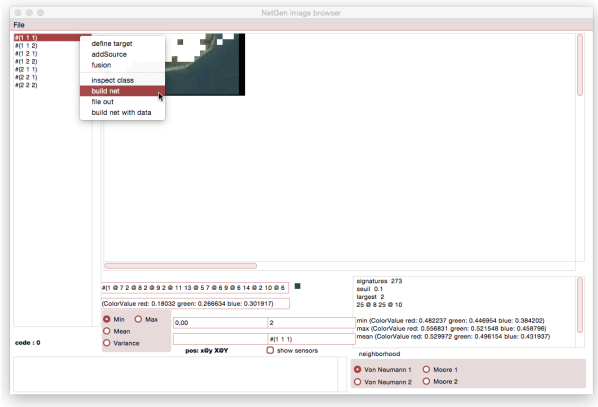

Figure 8.12: Calling the network generator on a cell class

8.2.2 Programs generation

Once the network is built, programs can be produced to represent its process and topology. Menu in the cell class list has a *Build net* option for this (figure 8.12. The results are produced in NetGen windows available as shown figure 8.13. There is atmost 4 processes in the fanout for each cell, and isolated process are removed. The abstract network is described as shown below :

```
cellNetwork0

messages none  defined.
Px23y8 { Px22y8, Px23y9, Px24y8, Px23y7 } CellNode (23 @ 8) (20)
Px20y10 { Px20y9, Px20y11, Px21y10, Px19y10 } CellNode (20 @ 10) (20)
Px17y7 { Px17y6, Px16y7, Px17y8, Px18y7 } CellNode (17 @ 7) (20)
Px12y8 { Px12y9, Px12y7, Px13y8, Px11y8 } CellNode (12 @ 8) (20)
...
```

In this text, each process is named using its position, as example *Px*23*y*8 is located at row 8, columns 23. The name of the procedure executed is CellNode() to distinguish with Node() used inside sensor networks. Positions and range also appear at the end of the line.

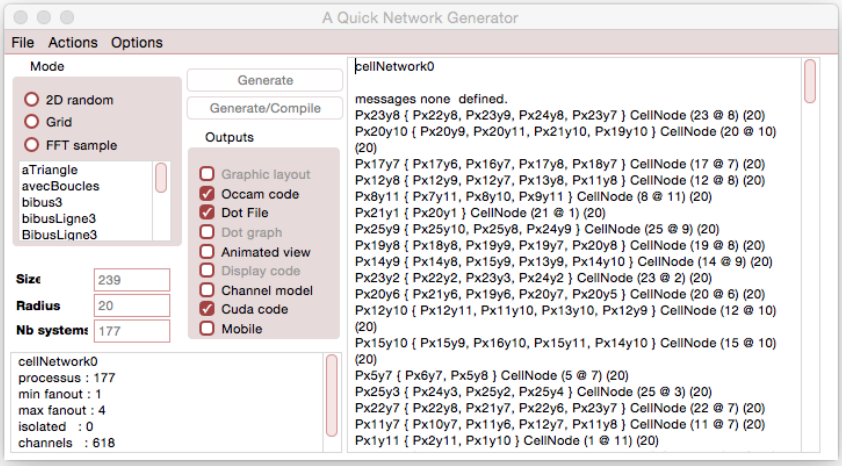

Figure 8.13: Screencast showing the abstract specification of a generated network for code 0 class. One process per line, as for sensor networks.

Statistics show that fanout is limited to four, fanin at least equal to 1. There are 177 cells. Occam (and CUDA) process systems have been produced by the generator that still need behaviour description for their cella. This behaviour can be very standard transition rule for cellular automata, but networking behaviuor could apply as well:

- diameter computation : longest distance between cells,
- leaders : for naming and differentiating subnetworks
- routing tables : to reach any cell from any cell in the same network
- packet transport and delivery, etc...

8.2.3 Compiling and simulating

The shortest path to simulate is to reuse a sensor network Occam behaviour file, as example for leader and diameter dynamic computation. The Actions menu of NetGen window allows to do it (figure 8.14).

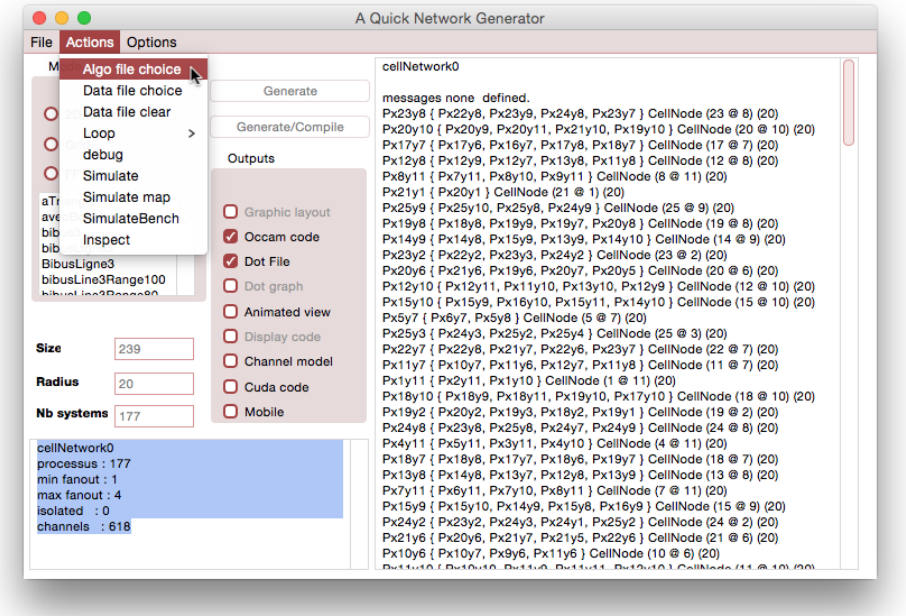

Figure 8.14: Selection of include files: for behaviour, and also for data files holding the cell contents.

Once this is done, the action flow is asfollow:

1. Check coherency between behaviour file and architecture file: in particular the name of the procedures executed by processes must be the same. For this example, it is CellNode(), defined in *cellnodes-test-include-diam.occ*.

2. Compile architecture. As example :
   kroc -lcourse cellNetwork17.occ

3. Check the execution trace.

8.3 Physical cell system illustration : Antsiranana geography capture and analysis

8.3.1 Sample : Antsiranana cell classification

We then select one pixel space and generate networks. and graph for neighborhood Von-Neumann 1 The graph seems to present a single network, isn't it?

8.3.2 Neighborhood statistics

The four standatd neighborhood were selected in turns. Statistics came from the NetGen window after accepting the different network (table 8.1).

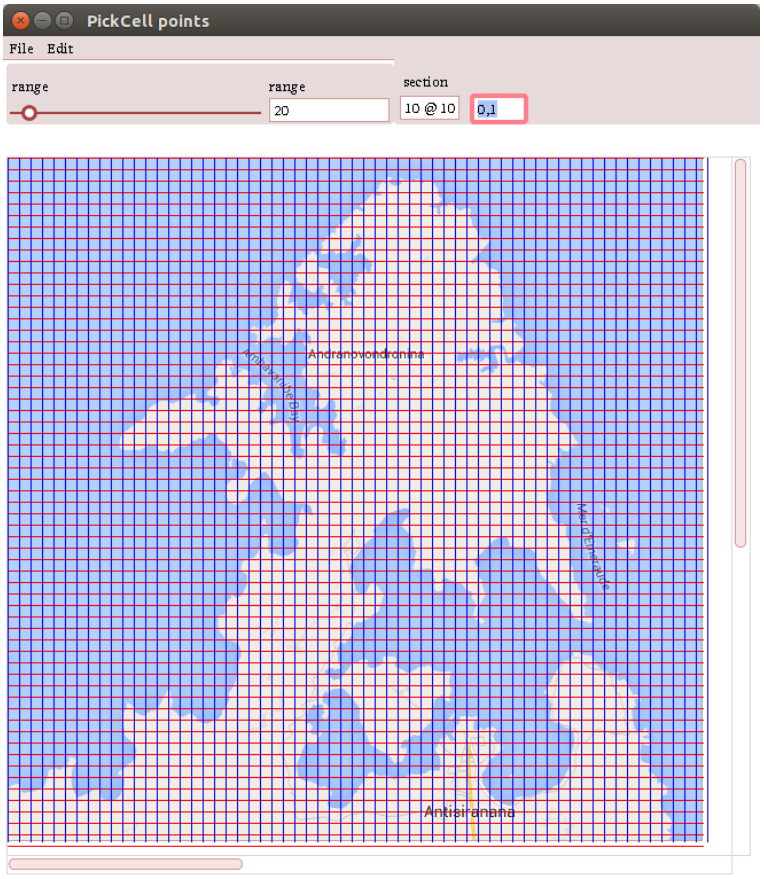

Figure 8.15: Google map API3 was used to select Antsiranana bay and feed PickCell : 10×10 pixel cells.

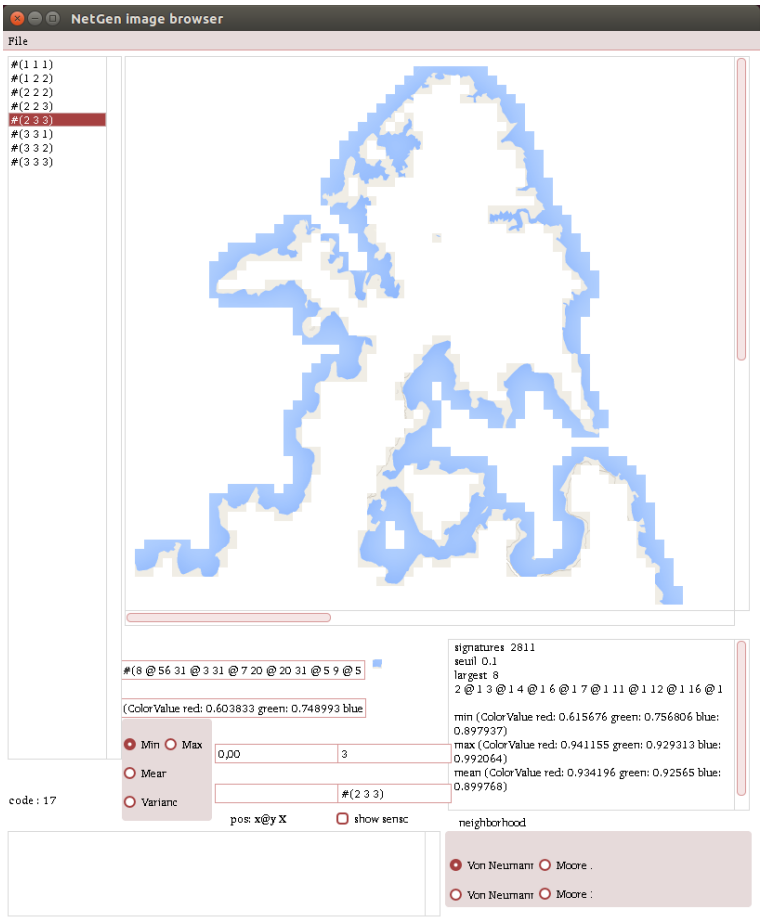

Figure 8.16: The pixel space was split into 27 subspaces defining classes.

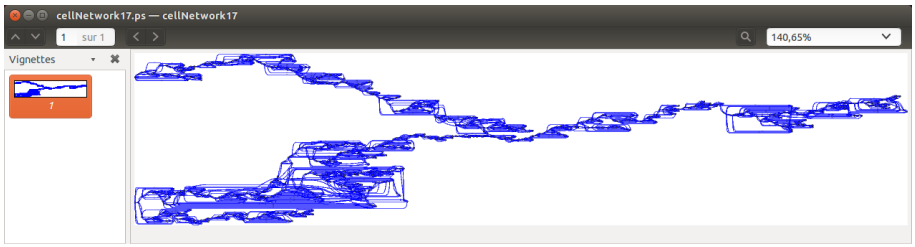

Figure 8.17: Resulting logic graph for Von Neumann1 neighborhod. How many subgraphs ? see section **??**

cellNetwork17	VN1	VN2	M1	M2
processus	809	809	809	809
min fanout	1	3	2	6
max fanout	4	12	8	22
isolated	0	0	0	0
channels	2440	6346	4576	10962
neighborhood size	4	12	8	22

Table 8.1: Four different neighborhood generated for Antsiranana bay with distances 1 and 2. the table show cell networks characteristics.

8.3.3 Trace and execution

Once the program is compiled, we can launch an execution, with more than 800 processes working together. The program compute the leader Id, and the diameter of network(s). The trace coming from execution has the following contents:

```
 Id     diam    leader
 784      84 803
   2      84 803
   5      84 803
...
 803      84 803
 668     197 808
   0     197 808
   1     197 808
...
```

identity	diameter	leader
784	84	803
2	84	803
5	84	803
...	..	..
803	84	803
668	197	808
0	197	808
1	197	808
...	..	..
806	197	808
807	197	808
808	197	808

Table 8.2: Simulation trace: smallest networks (84-803) finish first in the case of Occam execution. It is not the case for CUDA execution.

Therefore we have two *cell networks* of leaders 803 and 808, and diameters 84 and 197. Any information circulating in the second network will use at most 197 hops to reach a destination.

All the algorithms are not equivalent. See table 8.3 showing a comparison between A and running on a multicore i7 CPU. It is noticeable that B has a strong advantage and a higher level of effective parallelism (see the user row).

time	A	B
real	5m24.271s	0m9.843s
user	5m40.865s	0m28.074s
sys	0m0.686s	0m0.674s

Table 8.3: Two programs A and B executed on the same network. A is for naive leader and diameter computations. B is for diameter computation saving useless data transmission.

8.4 Neighborhood discovery and isotropy

As for computer networks, cellular automata built from analysis have irregularities:

- shape are produced from classes, and shapes have frontiers. That means that if a Von Neumann neighborhood was taken as a connectivity pattern, cells do not have complete neighborhood.
- another problem comes from this lack of completion: how can we guess the direction addressed by a link array ? Which link lead to where ?
  - in the physical world links do not really exist, but direction can be significant,
  - cellular automata can be isotropic, or anisotropic, making directions critical in transition rules.

Figure 8.16 display a shore obtained from a map system. As discussed in section 8.3.3 the cell network is distributed into two parts showing diameters as large as 197 cell paths. If we are to model the effect of a particular wind on surface tides, we need to know the direction available at each cell point.

Thus local connectivity discovery is needed. As for sensor networks, it is interesting to maintain connectivity knowledge dynamically, and thus, during simulation.

8.4.1 Neighbourhood data structures explained

The element of Occam coding below allows to see what will append. Basically there is a descriptor for each detected neighbour. These descriptors will be stored in a dynamic table, and a redefinition of link protocol allows to pass these tokens on communication links (abstract, of course).

```
DATA TYPE KnownNeighbour
  RECORD
    INT Id: -- node identity
    INT linkIndex:  -- index of link
    Location relativeLoc: -- xLoc yLoc record
:
DATA TYPE Neighbourhood
  RECORD
    INT limit: -- to manage table size
    [MaxFanOut] KnownNeighbour knownNeighbour:
:
PROTOCOL diam.proto
  CASE
    null; BYTE
    file; FileOfId
    int; INT
    location; KnownNeighbour -- for neighbour discovery
:
```

Initially each cell has its Id and absolute location stored in a KnownNeighbour record. The problem is to fill the Neighbourhood table with elements coming from connected cells.

8.4.2 Neighbourhood discovery

In our synchronous simulation frame work, this is a single exchange step followed by the local building of the neighbourhood table.

```
PROC GetNeighbourhood()
  KnownNeighbour me:
  [MaxFanOut] KnownNeighbour them:
  SEQ
    GetLocation() -- into Location variable loc
    me[Id] := Id  -- Cell Identity
    me[linkIndex] := -1
    me[relativeLoc] := loc -- absolute location
    PAR
      PAR j=0 FOR SIZE inChan
        inChan[j] ? CASE
          location ; them[j]   -- get tokens from neighbours
            SKIP
      PAR j=0 FOR SIZE outChan
        outChan[j] ! location ; me -- send our token
    ResetNeighbourhood(myNeighbours)
    SEQ i=0 FOR SIZE inChan
      SEQ
        them[i][linkIndex] := i --recall link index
        AddInNeighbourhood(loc, them[i],myNeighbours)
         -- add a neighbour and translate absolute to relative location
    DumpNeighbourhood(me, myNeighbours, toMux)  -- trace
:
```

8.4.3 Trace and comment

This code was added to previous sample, and few lines were extracted. During the discovery phasis, each node report its identity, its position, then the neighborhood table.

The elements extracted all refers to node 482 located at $x = 32, y = 53$. This node has a full VN1 neighborhood with cardinal directions N,E,S,W mapped to link $3,1,0,2$. North is relative coordinates $(0,1)$, West is $(-1,0)$. The coordinate system see the origin in $(0,0)$ being at top left of an image.

The name of this cell is Px32y53, and its network diameter is 197. Notice that cell of Id 479 has an incomplete neighbourhood reduced to 3 cells.

```
./cellNetwork17 | grep 482
248 32 54 { 3 0; 0, 1}{ 57 1; 1, 0}{ 245 2; -1, 0}{ 482 3; 0, -1}
306 33 53 { 57 0; 0, 1}{ 380 1; 0, -1}{ 482 2; -1, 0}{ 741 3; 1, 0}
479 31 53 { 245 0; 0, 1}{ 482 1; 1, 0}{ 564 2; 0, -1}
482 32 53 { 248 0; 0, 1}{ 306 1; 1, 0}{ 479 2; -1, 0}{ 575 3; 0, -1}
575 32 52 { 380 0; 1, 0}{ 482 1; 0, 1}{ 564 2; -1, 0}{ 580 3; 0, -1}
482 Px32y53:
482  197
```

We can interpret the trace according to table 8.4. and .

direction	vector	link index
N	(0, -1)	3
W	(-1 , 0)	2
S	(0, 1)	0
E	(1, 0)	1

Table 8.4: Node 482 relations between link index and cardinal directions.

direction	vector	link index
N	(0, -1)	2
S	(0, 1)	0
E	(1, 0)	1

Table 8.5: Node 479 relations between link index and cardinal directions. Uncomplete VN1 neighbourhood.

8.5 Accessing cell data

How we can obtain access to cell data produced during image analysis. Look inside the data file produced as shown figure s 8.14 and 8.12.

8.5.1 Data structures

The data structure appearing in top of the data file are as follows:

```
DATA TYPE ImageExtent -- cell dimension
  RECORD
    INT width:
    INT height:
:
DATA TYPE CellPosition
  RECORD
    INT x:
    INT y:
:
DATA TYPE RGBPixel -- RGB pixel
  RECORD
    BYTE red, green, blue:
:
-- generated description of one cell, size hard coded
DATA TYPE Depth24ByteArray  IS [ 100] RGBPixel:
-- this is a cell image contents with dimensions and contents
DATA TYPE CellImage
  RECORD
    ImageExtent  extent:
    Depth24ByteArray pixelArray:
:
-- this is a complete cell, with its position and data
DATA TYPE CellArray
  RECORD
    CellPosition position:
    CellImage image:
:
-- this is data for one class, 810 cells for this one
VAL [   810] CellArray  Cells IS  -- follows a table of cell contents
-- pages of values
```

8.5.2 From cell data to cellular automata local state

(to be continued)

8.6 Comments

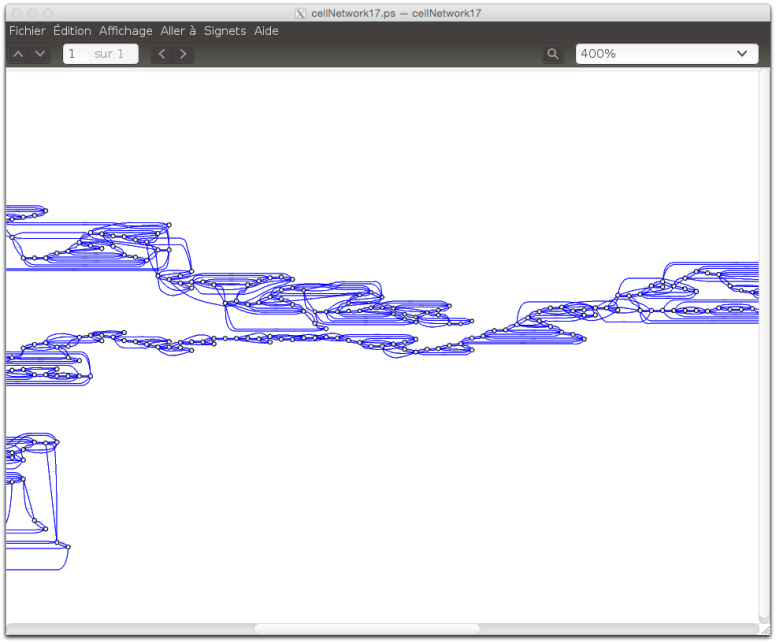

Figure 8.18: View on the logic graph..

Bibliography

[1] Ahmed Ahmed. Simulation et modélisation de diffusions physiques (in english). Master's thesis, UBO, Dept informatique, http://wsn.univ-brest.fr/TER-rapports/ahmedReport.pdf, 2013.

[2] Commission européenne. http://www.eea.europa.eu/themes/water/interactive/bathing/state-of-bathing-waters. 2013.

[3] http://Digi.com. Product manual: Xbee / xbee-pro zb rf modules. 2013.

[4] Van Nguyen Long Huu. A note on satellites and sensor communication frequencies. *Pacific Journal of Embedded Systems*, 34:541–545, 1970.

[5] Hiep Huynh. On languages with non-homogeneous strings of quantifiers. *Vietnam Journal of Mathematics and wireless communications*, 8:75–79, 1990.

Contents