Implementing Static Synchronus Sensor Fields Using NiMo *†

S. Clerici and A. Duch and C. Zoltan
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
{silvia,duch,zoltan}@lsi.upc.edu

April 25, 2009

Abstract

In this work we present some implementations of a *Static Synchronus Sensor Field* (SSSF) [2], a static and synchronus model for sensor networks in which a finite set of sensing devices are geographically distributed and can communicate among them. We used for implementations the NiMo (Nets in Motion) [1] programming language, a graphic-functional-dataflow language that allows a step by step visualization of the executions of an algorithm, making visible all the involved elements. Generator programs to build different topologies (rings, trees, etc.) of variable size produce the corresponding sensor networks in the first execution stage. Once it is generated, the network can be run in the next execution step. Experimenting with sensor networks gives an insight of their behaviour and aids to see their properties, as can be appreciated in this work.

1 Introduction

A *Static Synchronus Sensor Field* (SSSF) [2] is a static and synchronus model for sensor networks in which a finite set of sensing devices are geographically distributed and communicate among them. The communication scheme of such a network is captured by a directed *communication graph* $G = (V, E)$ whose set of vertices V represent the set of devices and whose set of arcs E the communication link of each pair of devices, each arc $(u, v) \in E$ indicates that device $u \in V$ can communicate (send information to) device $v \in V$.

Among the several problems that can be simulated and analyzed using the SSSF model, we focus our attention on the problem of *average monitoring* (AM) [2]. Given a SSSF of n devices taking each a measure at every step, the AM problem consists on produce, per each step and at each device simultaneously, the average of the n taken measures. The result will be produced with a certain delay (the latency of the system) that depends on G's topology.

*Partially supported by the ICT Program of the European Union under contract number 215270 (FRONTS). The second author was also supported by the spanish project TIN2006-11345 (ALINEX-2).

†This work is an extended version of the abstract presented in the 1st Workshop on New Challenges in Distributed Systems held in Valparaiso, Chile, April 6-9 2009.

We present then, some exemples of SSSF implementations solving the AM problem on different topologies and we use the NiMoToons tool (the NiMo environment) to show their execution. NiMo (networks in motion) [1] is a graphic-functional-dataflow language that allows a step by step visualization of the executions of an algorithm, making visible all the involved elements. The corresponding program is a network of processes and data and its execution exhibits its state at every instant.

Synchronization in NiMo is obtained by data availability, parallel execution and delays on channels of dataflow. This is in sharp contrast with synchronization obtained by memory allocation in each device, the way in which synchronization is achieved with implementations in an imperative programming language. In NiMo's implementations of the AM problem channels act as buffers and need to have a capacity proportional to the network size, but this is independent of the device and therefore free devices from memory constraints due to synchronization requirements.

Experimenting with sensor networks gives an insight of their behaviour and aids to see their properties. In particular, for SSSFs solving the AM problem, the NiMo implementation simplifies the verification of the programs correcteness, which is one of our main claims in this work.

The work is organized as follows. In Section 2 we give some preliminaries of NiMo programming language. We then show, in Section 3 the imperative solutions (introduced in [2]) and the NiMo implementations (object of this work) solving the AM problem in a SSSF under two topologies: an oriented ring and a complete and oriented binary tree. The implementations mentioned above consider that the size of the communication graph of the SSSF is fixed, but, in applications, even if the graph topology is known, its size can be variable, so in Section 4 we introduce the NiMo programs that generate, for both ring and tree topologies, a network of any given size. It is also possible that a given network calculates its own size as it is shown in Section 5. In Section 6 we show how NiMo's implicit parallelism is exploited in the referred programs. Finally, in Section 7, we give some final considerations and guidelines for future work.

2 The NiMo Programming Language

NiMo [1] is a general purpose typed graphical programming language designed to express common programming applications in a concise, elegant, and typesafe way. It implements the classical boxes and strings paradigm. In particular it is oriented to support stream programming and execution of open programs.

Its evaluation model supports setable behaviour for processes that range from lazy to week eager and exploits implicit parallelism. Its environment (NiMo Toons) turns out to be a workbench for program experimentation and provides special support for the construction, manipulation, and execution of NiMo code. Supporting a non-strict evaluation policy, NiMo is well suited to deal with infinite streams. It has a small set of primitives, including those needed for stream programming. For instance the *map* process is the generalization to n inputs and m outputs for the program scheme "apply to all". The *map* process includes a process parameter P that when applied to n inputs produce m outputs. The *map* behaviour is to apply P to each n incoming elements to produce a new result in each of its m output channels. An example of the *map* process can

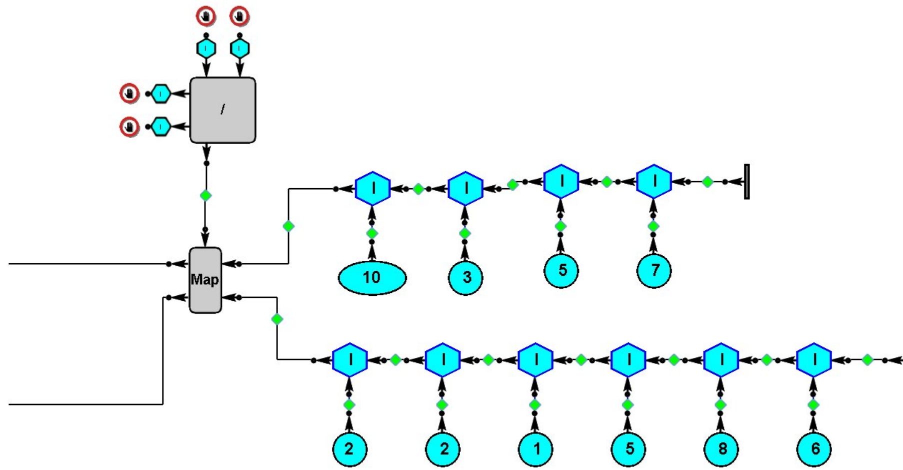

Figure 1: The *map* process with integer division

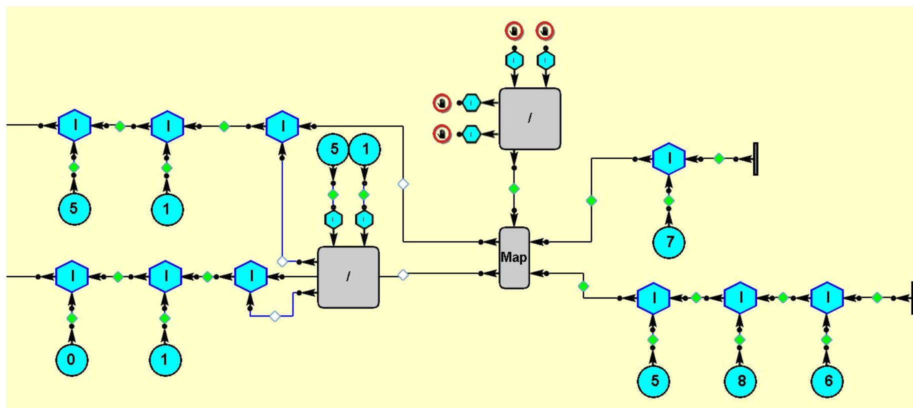

Figure 2: The *map* process after three execution steps

be appreciated in Figure 1, where we can see that *map* gives the quotient and the remainder of integer division componentwise. After three execution steps, in Figure 2, we can see in the output channel, two already calculated values and the third that will be evaluated in the next execution step.

Other useful stream processes are for example *SplitAt* or *SplitCond*. The first one, given an integer n, splits the input channel into two channels, the first of size n and the second with the remaining elements. The second, *SplitCond*, separates a list into two lists: the first holds all the values in the input channel satisfying the condition and the second holds the complement. Basic processes are combined to define a component, i.e., a new process that can be used later in programs or to define new components. A component has an associated *interfase* which is a white box (basic processes have grey boxes) with typed input and output ports.

3 SSSFs Implementations for the AM Problem

In order to solve sensor networks problems it is important to take into account (and try to optimize) parameters such as the latency of the network (elapsed number of steps between the first taken measure and the first computed average), the number of sent data items per device per step, the execution time and memory requirements per device per step, the amount of energy requirements, among others.

Unfortunately, in many occasions an improvement in some measures (like the latency) implies a lost in other (like the size of the sent messages). In particular for sensor networks there are many trade-offs among the different complexity measures.

Although there exists a generic algorithm for solving the AM problem in any SSSF with optimal latency (which is the diameter of the SSSF's communication graph[1] [2]), the a priori knowledge of the communication graph topology can considerably improve the algorithms that control the execution of devices, optimizing some of the complexity measures involved.

In what follows, we will show the imperative implementations introduced in [2] as well as their NiMo versions. It is the case of algorithms that minimize either the latency or the message length in SSSFs solving the AM problem. We focus on two topologies: oriented rings and complete binary trees.

3.1 Ring topology

In a SSSF whose communication graph is an oriented ring, each sensing device reads (measures) data from the environment, receives data from its predecessor (unique because of the topology), sends data to the environment and, sends data to its successor. We will now introduce two different implementations, the first one has optimal latency, the second one improves the message length (incrementing the latency) at every computational step (see [2] for formal definition of computational step under this model, but, informally, it goes from the *receive* up to the *send* actions–both included– of the synchronized devices).

3.1.1 Optimal latency

In order to be able to compute the average of the measurements taken by all devices in the network at a given time, a single device requires all these values, and, the smallest number of steps in which it can receive these values is the number of steps required by the value of its farthest device to arrive, in this case, $n-1$ steps (the diameter of the graph), considering that the ring consists of n devices.

In [2] a solution that has a latency of $n-1$ steps (optimal) in rings of n devices is introduced. In this particular solution, all sensors have a similar behaviour (parameterized by the size of the network), each transmitting, at each step, an amount of data proportional to the number of sensors in the network. In fact, a total of $n-1$ data items per step are trasmitted between neighbor sensors. If we call u_i^t the value read by the sensor i at step t, then the values transmitted between sensor k and sensor $(k+1) \mod n$ at step t is a tuple of

[1]The diameter $diam(G)$ of a graph G is the maximum distantace over all pair of vertices in a graph.

values $(u_k^t, u_{k-1}^{t-1}, \ldots, u_{k-n+1}^{t-n+1})$ which corresponds to the values read by the $n-1$ other sensors in the $n-1$ previous steps.

Imperative solution. The imperative algorithm executed by sensor k introduced in [2] is Algorithm 1. Since we are considering that the network has n sensors connected into an ordered ring, we have that every sensor k is connected to sensor $(k+1) \mod n$.

Algorithm 1 Optimal latency in a ring topology

```
// Algorithm for sensor k

// Initially
A[1, ... ,n] = [0.0, ... , 0.0];
X[1, ... ,n-1] = [0.0, ... , 0.0];

// Step

// receive
receive X;
receive input data u;

// compute
for (i = n-1; i >= 1; --i)  A[i+1] = A[i]+X[i];
A[1] = u;
for (i = n-1; i >= 2; --i) X[i] = X[i-1];
X[1]=u;
v= A[n]/n;

// send and output
send X;
output v;
```

As we already said, at every step, sensor k receives information from sensor $(k-1) \mod n$ and sends information to sensor $(k+1) \mod n$. It is assumed that each sensor knows the number n of sensors in the ring, but, if it were not the case, in this topology it could be calculated easily with a corresponding increment in latency (see Section 5).

For each sensor k the algorithm considers a vector X of size $n-1$ that stores the tuple $(u_{k-1}^{t-1}, \ldots, u_{k-n+1}^{t-n+1})$ comming from sensor $(k-1) \mod n$. X stores also the tuple of data items that will be send to sensor $(k+1) \mod n$. There is an additional vector A of size n that stores partial sums.

At step t, A has stored in its first position the measure taken by the sensor (k) at previous step $(t-1)$, in its second position the sum of its measure and the one of its predecessor in step $t-2$, and so on up to the n-th position where it has stored the sum of the measures taken at step $t-n$ by all sensors. The value of this n-th position (divided by n) is the value that was sent by sensor k to the environment at previous step $t-1$.

When vector X is received, A is actualized as follows. The n-th position of A

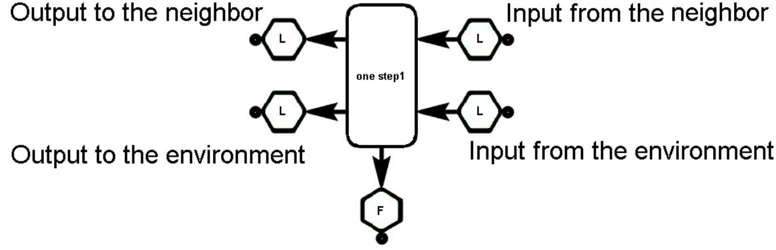

Figure 3: Sensor interfase

is the sum of its $n-1$-th position and the $n-1$-th position of X, it corresponds to the sum of the $n-1$ devices at step $t-n+1$ and is the value that will be used to send the average to the environment at this step. The rest of positions are actualized in a similar way, up to the first one wich is equal to the data item received from environment at this step. The values of X are moved one position to the left and the first one takes the value of the data item comming from the environment at this step. Finally X is sent to next device.

Using this algorithm, the Average Monitoring problem can be solved with an optimal latency of $n-1$ steps. At each step, device k take a measure, receives $n-1$ data items from its predecessor corresponding to measures taken by device at distance i (for $i \in \{1, \ldots, n-1\}$) at the i-th previous step, computes and outputs the average of the $(n-1)$th step and sends to its successor the $n-1$ corresponding measures (the $n-2$ received from its predecessor plus its new taken one). Hence, the complexity measures are as follows, the execution time $T(n)$ per device per step is $T(n) = O(n)$, the space requirements per device per step is $S(n) = \Theta(n)$, the message number sent per each device at each step is $MN(n) = 1$ and the corresponding message length is $ML(n) = n-1$.

We now explain how to implement Algorithm 1 in NiMo.

NiMo Solution. The solution is based on a set of identical components that we called *onestep*, the interfase is given in Figure 3. It receives 2 input data streams and produces 2 output data streams.

Using this component a network is built having ring topology, with the desired number of sensors (in our example case will be three).

Data flows from right to left of the screen, and each sensor's output is connected to an hexagon as shown in Figure 4. In channels that connect two sensors we can see the values (in this case a two-elements list and in the general case an $(n-1)$-elements list, initialized to zero) that flow from one sensor to its neighbor. The second input in each sensor is left open to be connected with the environment.

Whenever the ports are connected to the environment in the corresponding location, the sensor will take measures from the environment.

In Figure 5 we see the sensor program. Input values (incoming arrows having a numbered yellow box on it's rigth), numbered 1 and 2 (in the example), are used for two purposes, therefore they are duplicated via the black dot process. The first set of copies serve to calculate the value that should be output to the environment and the values that should be saved for the next interaction,

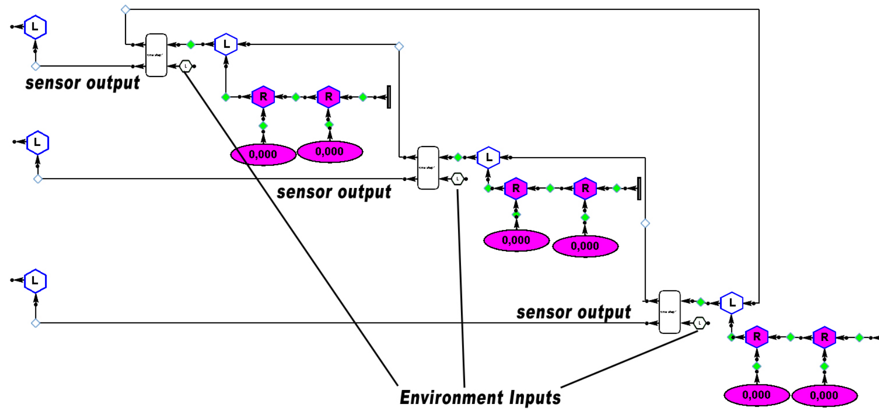

Figure 4: Example: a three-sensor network

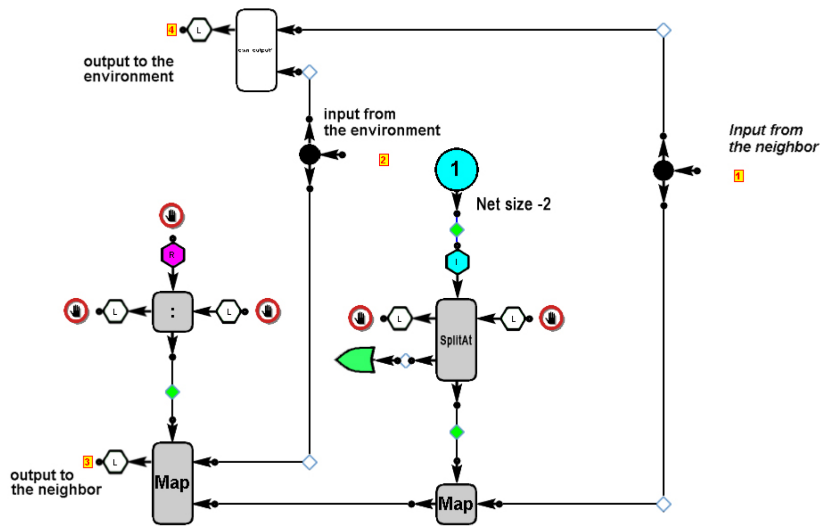

Figure 5: A sensor

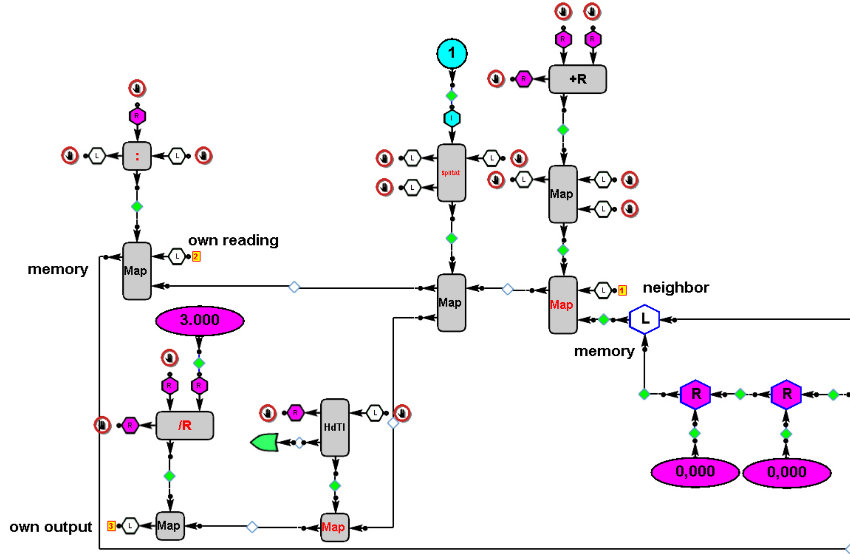

Figure 6: Sensor: output to the environment

so become the input to the process expanded in Figure 6. The second set of copies serve to construct a new value for the neighboring sensor. This value is constructed by dropping the last received value in the list coming from the neighbor and prefixing to the list the newly received value from the environment.

Dropping the last element of the list is done by using the *SplitAt* process. In our example, a network with 3 components, it is set to 1. Since this action is to be done to every list sent by the neighbor, the *SplitAt* is "stream-ized" by the use of the *map* proccess that applies its function parameter to each incoming element. The output will be a list of one element ($n-2$ elements in the general case), which feeds another *map* process that constructs via the process "*:*" a list of two elements ($n-1$ in the general case), with the element read from the environment and the remaining values that came from the neighbor.

In Figure 6 we see a list of values painted in magenta. We will call these values "memory" because the list circulates in a circuit that always contains exactly one list. This list holds the partial sums of the readings of all other sensors in the network and will be added componentwise to the list of the rest of sensors, coming from the neighbor.

The last item in the list will contain the sum of all sensor readings in a step. Therefore, this last value will be a result to the environment (yellow rectangle numbered 1) and will be removed from the "memory". As this "memory" has size equal to $n-1$, the sensor parameter must be $n-2$, which in our 3-sensor network is 1. The new "memory" builds on the last one by prefixing (using the process *:*) the last reading of the sensor. And since the same operation needs to be executed for every "memory", the *map* process is used for transforming it.

Execution is synchronized by data availability and the cost of obtaining a result depends on the calculation of $n-1$ sums. NiMo exploits as much as possible parallel execution, but as it is a stream oriented language, it does not have yet facilities for treating parallel vector operations, a capability that would

be useful in this case.

In NiMo a computation step is the expansion/execution of all processes in the network that are able to do so. Therefore, in this solution, that mimics the imperative algorithm, the number of steps (steps of the model multiplied by execution time) needed for obtaining a result depends on the size of the network and is summarized in the following lemma.

Lemma 3.1 *Each channel in the circuit has at most one list consisting of $n-1$ elements. A value goes to the output every n steps.*

3.1.2 Improving the message length

Imperative solution. By data aggregation and allowing a larger latency, it is possible to improve the size of each sent message as we show below.

Considering again a communication graph of n devices under an oriented ring topology, Algorithm 2 solves the AM problem with messages of smaller size (at most the sum of n values), at each step.

Informally, one device acts as a leader (say device 1) and another device (say device n) computes, collects and distributes the averages. It is assumed that each sensor knows the number of sensors n as well as its position in the ring. As before, all the nodes start reading from the environment at the same time.

Device 1 takes and flows its first taken measure to device 2 at step 1. At step 2, device 2 receives the first taken measure of device 1, adds it to its own taken measure at step 1 and forwards the sum to device 3. Eventually, sensor n receives the sum of measures taken by devices $1, 2, \ldots, n-1$ at step 1, adds it to its own taken measure at step 1, computes the first meaningful average and forwards it to all other devices.

NiMo Solution. Another solution to the problem of gathering inputs from all the sensors to produce an identical result at each sensor at the same step, can be solved by having a stream of partial sums, collecting readings at the same step. A value in this stream, entering a given sensor, is added to the value read by the sensor at the corresponding step. Synchronization is obtained because a buffer of allready read values keeps them until the corresponding partial sum arrives.

The solution is based on a distinguished sensor and a set of generic sensors as we see in Figure 7. The generic sensor has two subnetworks. The one on the top adds the value from the environment (yellow square numbered 2), to the partial sum comming from the neighbor (yellow square numbered 3) and sends the result to the neighbor. The second subnetwork receives a partial sum of sensor readings in a given step and passes it to its neighbor, delaying a copy of the value before sending it to the environment. The number of steps that the output needs to be delayed depends on the position of the sensor in the network. This value is the sensor parameter (yellow square numbered 1), which is also the parameter to the component *delay2*.

The component interfase for these class of sensors has one parameter, three channel inputs and three channel outputs.

The distinguished sensor is the one that initiates the flow of the value to the environment at each sensor. This sensor does almost the same work as the other sensors, but as it does not receive a flowing value, so it has less input ports.

Algorithm 2 Algorithms that improve the message length

```
// Algorithm for sensor 1
// Initially
D[1] = [0.0]; A[1, ... , n-1]=[0.0, ... ,0.0];
// Step
// receive and input
receive  avg;
receive input data u;
// compute
D[1] = u;
sum = D[1];
for (i = n-1; i >= 2; --i) A[i] = A[i-1];
A[1] = avg;
// send and output
send sum, A[1]; output  A[n-1];

// Algorithm for sensor k
// Initially
D[1,...,k]=[0.0,...,0.0]; A[1,...,n-k]=[0.0,...,0.0];
// Step
// receive and input
receive sum, avg;
receive input data u;
// compute
for (i = k; i >= 2; --i) D[i] = D[i-1];
D[1] = u;
sum = sum + D[k];
for (i = n-k; i >= 2; --i) A[i] = A[i-1];
A[1] = avg;
// send and output
if  (k == n-1) send sum; else send sum, A[1];
output  A[n-k];

// Algorithm for sensor n
// Initially
D[1,...,n]=[0.0,...,0.0]; A[1,...,n]=[0.0,...,0.0];
// Step
// receive and input
receive sum;
receive input data u;
// compute
sum = sum + D[n];
for (i=n; i >= 2; --i) {D[i]=D[i-1]; A[i]=A[i-1];}
A[1] = sum / n;
D[1] = u;
// send and output
send A[1]; output A[n];
```

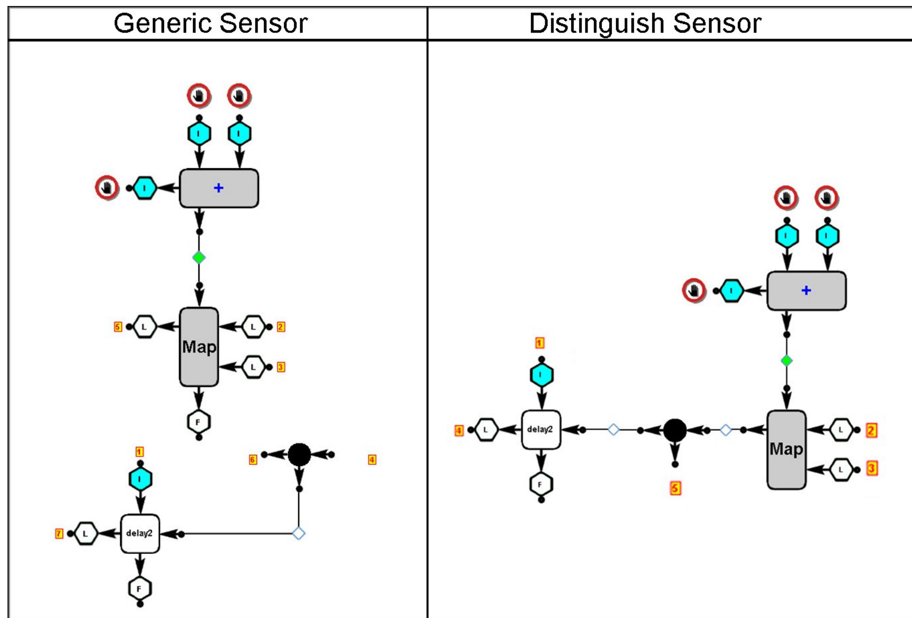

Figure 7: Two kinds of sensors for the ring solution

It also has a single value, the one that adds the value from the environment to the partial sum, comming from the neighbor. Now the result is a total sum, therefore the value goes to the neighbor after being duplicated (black dot) and enters the process *delay2* to be delayed (n steps) before going out to the environment, waiting for all other sensors to receive the value that has to be outputed at each location.

Figure 8 shows a network with four sensors, three of them having three inputs and three outputs and the sensor on the top, which is the distinguished one (Figure 7). The sensor that provides the distinguished one, has an open output port: the one that sends the flowing value, because that value does not need to return to *sensor0*. Since the sensor provided by the distinguished sensor, is the one that starts the partial sum of readings, it has allways a zero as initial value for the sum. Therefore, the port that is used in other sensors for connecting to the provider sensor is conected to a process that generates a stream of zeros (*repe* process).

In Figure 9 a three sensor network starts its execution. The sensor's input ports are connected to a possibly infinite sequence of values. The sensor on the top will have as inputs from the environment the values $0, 1, 2, 3, \ldots$, the second one the values $10, 11, 12, \ldots$ and the third one the values $20, 21, 22, \ldots$ so the resultant sum must be $30, 33, 36, \ldots$ at each sensor.

In Figure 10 we see a 3-sensor network executing. The 3-sensor network has produced one value (30) in each sensor, on the output port to the environment. At the top output channel there are two values (33 and 36) in place waiting to be sent to the environment. The value 36 is duplicated due to the red rectangle around the duplicator, and sent to the channel to start flowing the values to other sensors.

Looking at the addition processes (+), there are two marked with red rect-

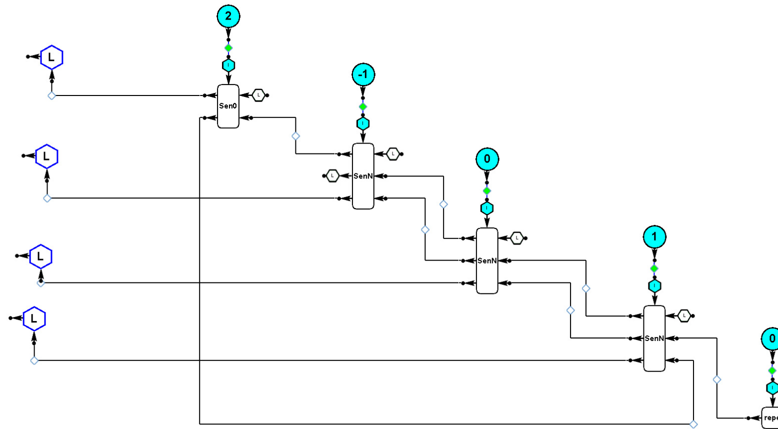

Figure 8: A 4-sensor network with open input ports

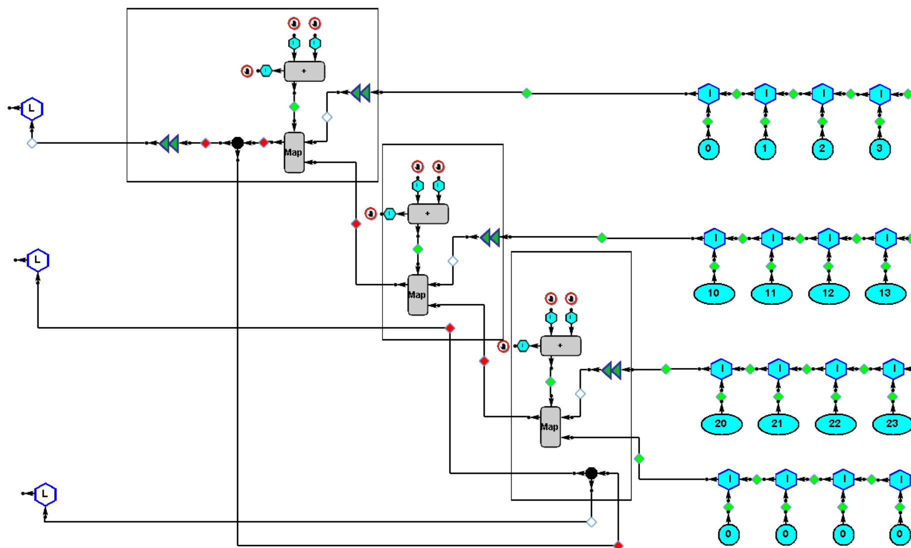

Figure 9: A 3-sensor network before execution

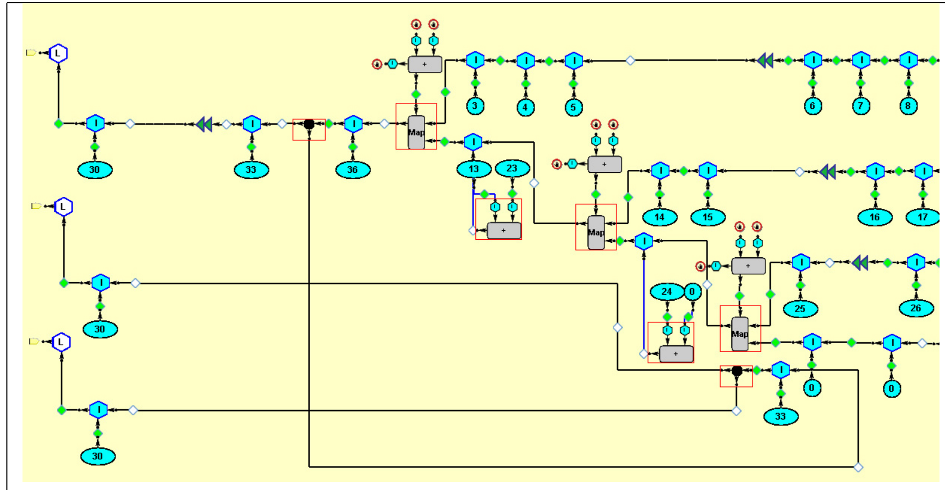

Figure 10: The network execution after producing the first value

angles, so are ready to act. One next to the bottom of the figure, that adds 0 to 24 (the reading at step 5 by Sensor 2). The other one will add the readings (23 and 13) at time 4 of Sensors 2 and 1. There is also a *map* process marked with a red rectangle. This process will apply its parameter (+) to the pair of incoming values: the partial sum at step four $(13 + 23)$ and the own reading at step four (3) that will produce the next value (9) to the environment[2].

Looking at the green double triangles on the rigth of the figure, triangles that belong to sensors, every sensor reads a value from the environment at step seven (the one on the top reads the value 6, the next reads the value 16 and the sensor on the bottom reads 26). Channels between the process *map* and the green triangles are the buffers for each sensor. Let's note that the first sensor has three values (the ones read in steps 4 to 6), the second has already consumed the value read in step 4, and the last sensor has only the value read in the previous step.

In Figure 10 there are several processes ready to be evaluated in paralell. This NiMo characteristic allows to simulate the network behaviour of each sensor acting on its own. In simulations, we see that the output to the environment keeps the same speed as the input from the environment. This is acheived due to NiMo's parallel execution.

3.2 Complete binary tree topology

Optimal latency. In a complete binary tree topology of the sensor network, each sensor has a very simple logic: it adds the readings from its two children with its own reading and sends the result to the father. Partial sums are therefore collected until the root gathers the total sum. All internal vertices need to flow the value to all sensors and does so by sending it to their children. The leaves output the value as soon as it is received, whilst the internal sensors need to delay their outputs depending on its height in the tree. This delay is equal to

[2] In the example we will use integers and obtain the sum of all sensor readings in order to simplify the presentation

the distance of the node to one of its leaves. The imperative program is given by Algorithm 3.

Algorithm 3 Algorithm for complete binary trees

```
// Algorithm for sensor k

// Initially
const h = log(n+1) - int(log(k)) -1;
up[1, ... ,h] = [0.0, ... , 0.0];
down[1, ... ,h] = [0.0, ... , 0.0];

// Step
// receive and input
receive input data u;
if (k != 1) receive y; // from father
if (k<=n/2) receive x1,x2; // from children
// compute
if (k > n/2) X = u;
else X = x1 + x2 + up[h];
for (i = h-1; i > 1; --i) up[i+1] = up[i];
up[1] = u;
if (k == 1) Y = X;
else Y = y;
v = down[h]/n;
for(i = h-1; i > 1; --i) down[i+1] = down[i];
down[1] = Y;
// send and output
output v;
if (k != 1) send X; // to father
if (k <= n/2) send Y; // to children
```

In the NiMo program there are three kinds of nodes in a sensor network with a complete tree topolology for solving the AM problem. All the nodes have their own channel for readings from the environment. Every sensor, but the root, have an input to collect the value to send to the output. Every non-leaf sensor have two inputs comming from each son. The values received from the sons, and its own reading are added in order to provide the value to its father. A sensor, to be able to do the operation, must keep values from the proceeding time intervals. Therefore the root will need a channel with capacity $\lceil \lg n \rceil$.

In Figure 11 the three kinds of sensors are displayed and Figure 12 shows a sensor network having a complete tree topology of depth three. The network was generated by means of a generator program as it is explained in next section. Running this network the first result is produced at all the sensor outputs at step 8, an then, a new result is produced at each step.

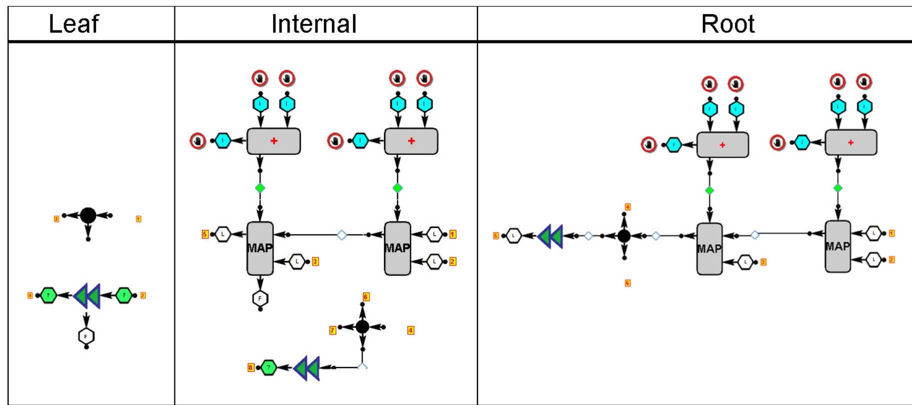

Figure 11: Tree sensors

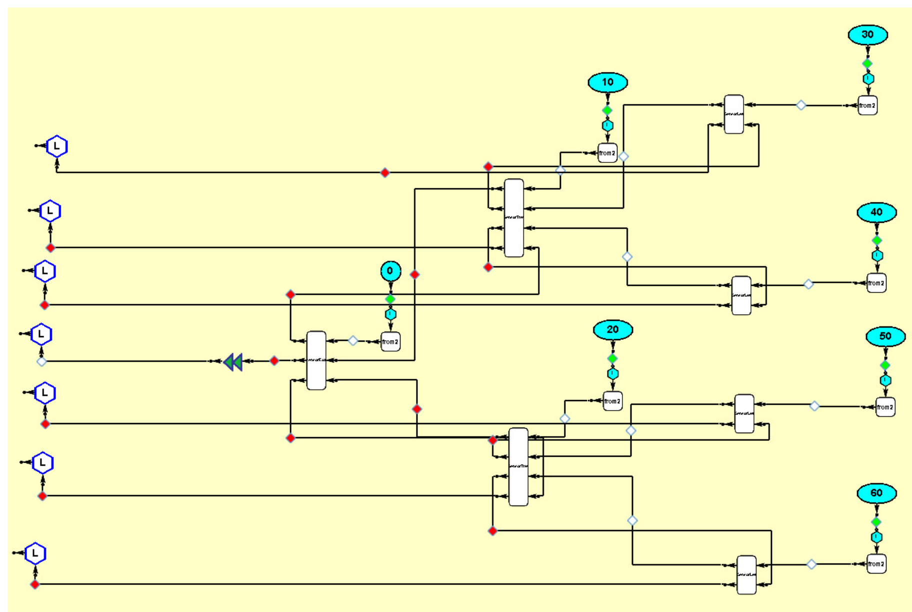

Figure 12: A tree sensor network

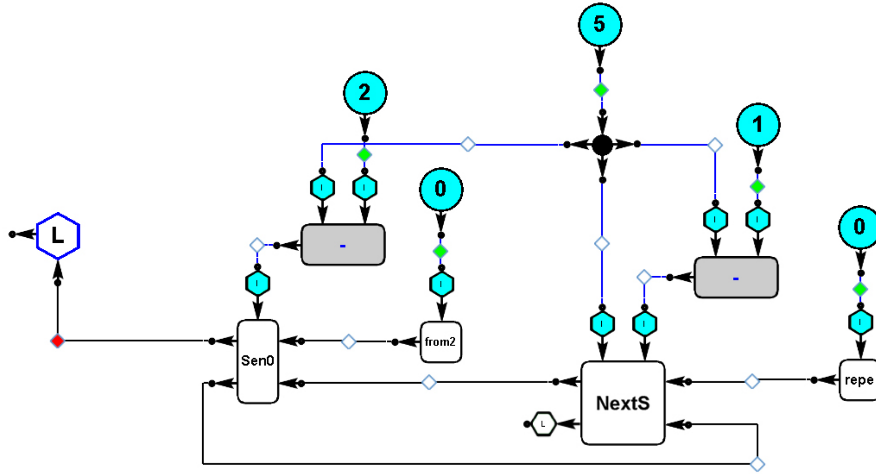

Figure 13: The ring topology generator

4 Generating static synchronous sensor fields

In this section we show how NiMo multi-stage programming (MSP [3]) characteristics are used for the generation of different network sizes and topologies.

In addition of having the usual constructs of a general-purpose language, multi-stage languages internalize the notion of runtime program generation and execution. Thus, multi-stage languages provide the programmer with the essence of partial evaluation and program specialization techniques, both of which have shown to lead to dramatic resource-utilization gains in a wide range of applications. In particular this technique is useful to reduce computation steps, in order to synchronize execution.

When the program has a regular structure, which is the case for sensor networks in the network topologies presented earlier, a NiMo program can be designed such that at the beginning of the execution it generates the sensor network of a given size. Once the network is generated, the execution becomes the network simulation.

Ring topology. Networks of ring topology with two kinds of sensors as the one in Figure 8 can be generated by means of the program shown in Figure 13. It has the distinguished sensor *Sen0* connected to a component process *NextS* that recursively generate the others $n-1$ sensors according to a ring topology. The network parameter is the number of sensors (5 in the example). Each sensor is parameterized by the number of steps that it has to delay its output (see the case for four sensors in figure 14). Each newly generated sensor is the provider to the previously generated sensor. The last generated sensor will be provided by *Sen0*.

All sensors included in this construction will have their execution mode set to *disable*, to inforce that no sensor will execute until the network is completely constructed. Also, in the construction, there are included processes that simulate the input from the environment for each process in the network. To prevent screen population the processes generating values to simulate the environment

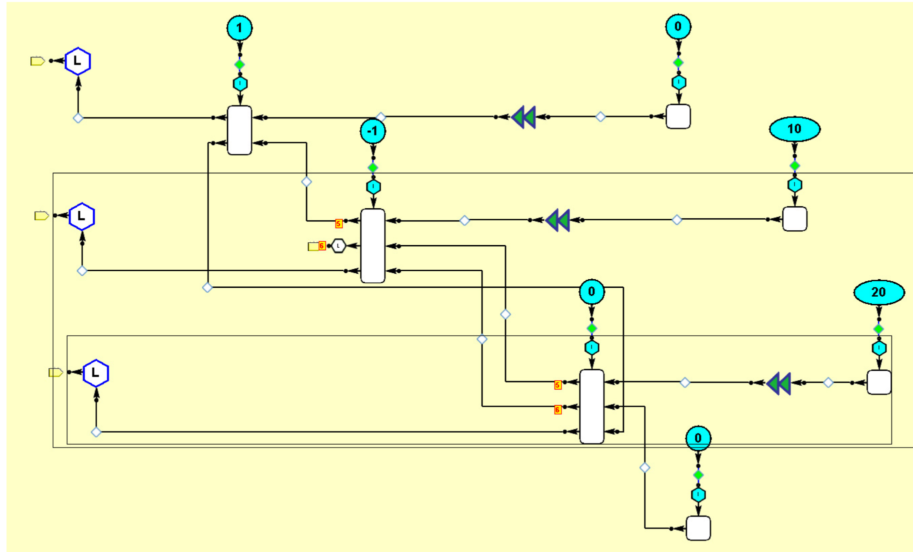

Figure 14: A generated 4-sensor network

are also set to *disable*.

When the execution stops, the network is already constructed. Using the NiMo Toons command that allows to globally change the processes execution mode in a network, all processes on the screen change their execution mode to *data driven*, therefore, every sensor will be expanded and replaced by its program at the same time. Once generated (as seen in Figure 14), experimentation (simulation) using NiMo Toons can be performed. The network execution allows to know how many steps it takes to produce the first meaningful value, how many steps between consecutive outputs, how many processors were busy at each instant, etc. Generating networks of different sizes is just to change a single parameter. In the network of Figure 13, the value is 5.

Complete binary tree topology. As was the case for ring topologies, a NiMo program can be designed to construct a regular tree network, having the height of the tree as a parameter.

In Figure 15 the generating program is shown for a tree with depth 3 (the root plus the two following levels). The generating program has the sensor Root connected to two instances of the component NextT (with parameter 2), which together generate recursively the following levels (n-1 levels of intermediate sensors and the last level of leaf sensors). As we can see in figure 12, the construction phase produce the non linear structure of the complete tree network. Also in this case, different sequences of input values to simulate the environment data are generated for each sensor.

5 Calculating the size of the network

In Sections 3.1.1 and 3.1.2 solutions were based on the knowledge of the size of the network and the relative position of the sensor in the network, these

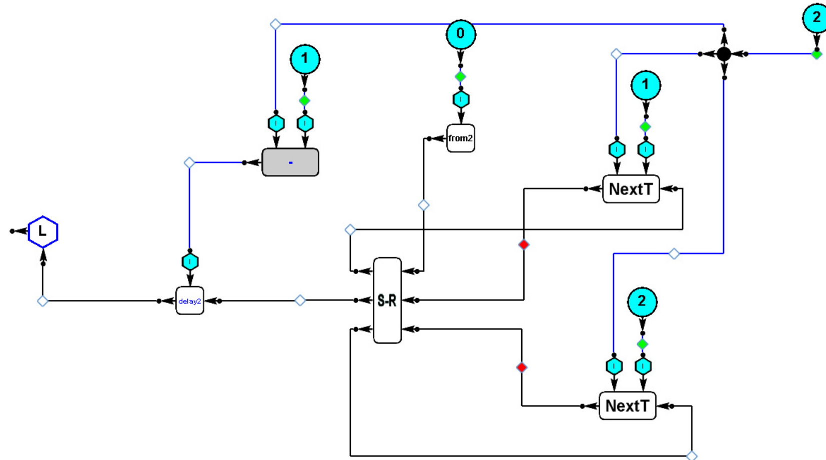

Figure 15: NiMo program for generating a complete tree network

values were a parameter of each sensor. Using NiMo capability of run-time specialization, network size can be calculated and used at the beginning of the execution to specialize each sensor program. This cost is assumed as an initial system stabilization. In Figure 16, instead of the parameter with the sensor position, an additional input channel is added to do initialization, for both kinds of sensors, and also an additional output for the rest of sensors except the distinguished one . The sensor provided by the distinguished sensor will have an input channel with a single element one.

The last observation alerts that it is necessary to identify the distinguished sensor and the neighbor.

In order to circumvent this difficulty, the distinguished sensor could include the initialization of its neighbor as the first one in the network, sending the value one to the neighbor. In this configuration, there is a distinguished sensor and all the other sensors receiving a value from the neighbor, using it as its position value and passing this value plus one to its corresponding neighbor. By run time specialization, this additional channel becomes useless after a few computing steps.

6 Parallel execution

In Figure 17 we can see a 5-sensor network with ring topology. After twelve computation steps, three values are already produced into every output channel, there are several processes surrounded by a red rectangle, meaning that that process will act in the step. In the figure we see a small window that shows graphically how many processors are used in each computation step. As we see in the curve, the tendency is to stabilize in 45 processors acting in a 5-sensor network. Parallelization allows the output frequency to be the same than the input frequency.

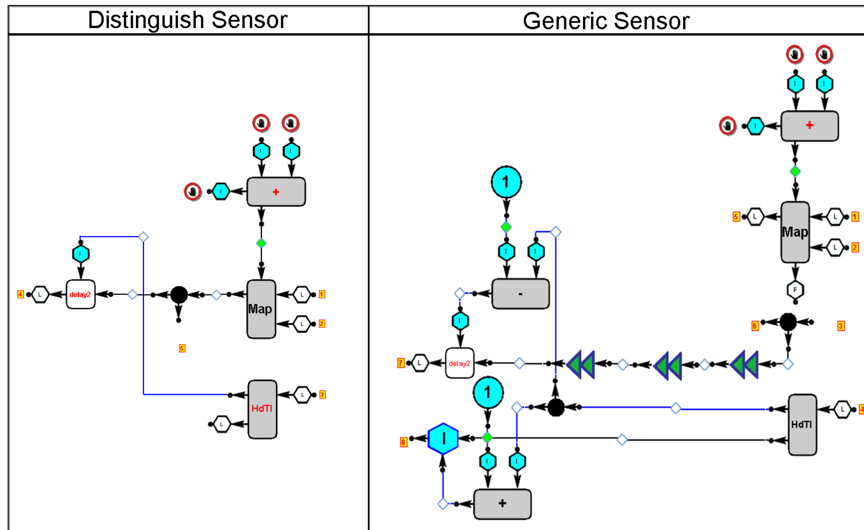

Figure 16: Sensors calculating their position

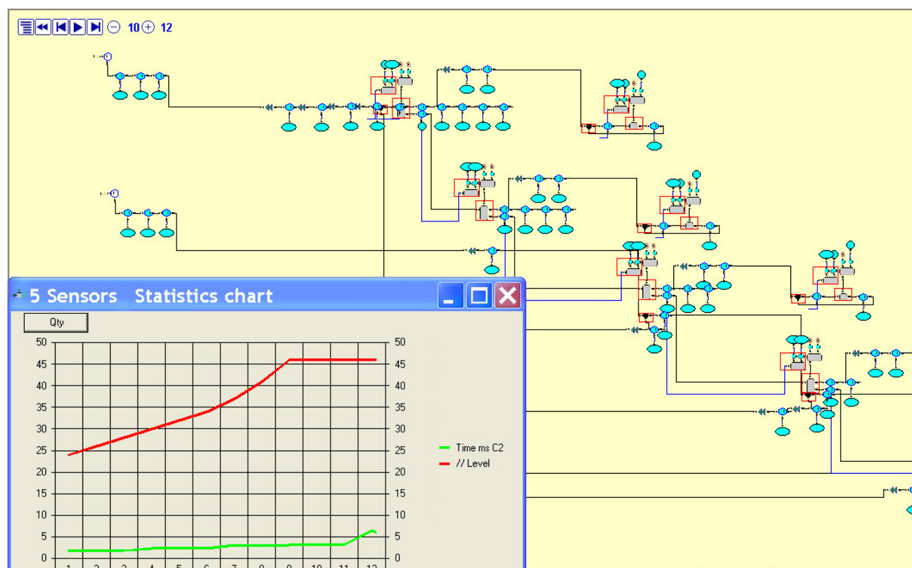

Figure 17: Number of processes used

7 Conclusions and future work

The work developed up to now shows the feasability of NiMo as simulation language for networks with sensing devices and its usefulness for experimenting on this kind of applications. Sensors are relatively simple engines that can be easily programmed using the NiMo primitives because the problem has a natural stream programming solution.

Channels have a queue behaviour that avoids the need to represent queues as vectors as it is done in the imperative solution. Channels also act as internal buffers for the sensors. That makes the code shorter and simpler. In addition, NiMo programs exhibit an implicit parallelism that can be exploited. Experimenting with networks of different sizes or topologies allows to verify hypothesis on the networks' behaviour. Therefore the language multi-staging facilities is quite useful in order to mechanically generate different networks that can start simulation in the next execution step. As future work we are interested in testing the NiMo stream language in other problem domains as mobile and self-stabilizing systems.

References

[1] *A graphic functional-dataflow language.* S. Clerici and C. Zoltan, Trends in Functional Programming, Intellect, 2006.

[2] *Average monitoring and alerting in static synchronus sensor field.* C. Àlvarez, A. Duch, J. Gabarro and M.J. Serna, personal communication, 2009.

[3] *Semantics, Applications, and Implementation of Program Generation* Taha, Walid.,J. Funct. Program. (13) 3, p. 453–454, Cambridge University Press, NY, USA, 2003.