A Process Oriented Development Flow for Wireless Sensor Networks

Guillaume Kremer Jimmy Osmont
CS Dept., UBO
France
g.kremer@wanadoo.fr
Jimmy.Osmont@univ-brest.fr

Bernard Pottier
LabSTICC UMR CNRS 3192, UBO
France
pottier@univ-brest.fr

Abstract

To ease the development for sensor networks, we propose a two-stage top-down flow: design and simulation, then synthesis of sensor code. The flow is an alternative to a direct design at node level.

The first operation is to describe network topologies on a graph model using textual or interactive tools. Simulations are achieved by generating a system of processes into an Occam program which is compiled and executed. This program represents a distributed machine conforming to the synchronous communication model, which is a strong reference for algorithm design.

The network simulation allows an early check of solutions to problems such as routing or dumping network data in various deployment topologies. It provides fast exploration of algorithm, and topology spaces, focusing on the difficult collective behaviour rather than local level programming.

1. Introduction

Wireless Sensor Networks (WSN) is an emerging domain with numerous applications arising, that take benefits from the communication and sensor technology progresses. Typical nodes for WSN are very small systems grouping a microcontroller, a wireless communication transceiver, local acquisition and control subsystem, and a power supply. Node hardware size can be in the order of few square centimeters with very low costs, the technology tendency being still to decrease node sizes and power consumption by integrating memory, logic and analogue interfaces on *mixed-signal* circuits[Doboli and Currie 2007].

IWST '09 September 2009, Brest.

WSN organizations can vary a lot, grouping few nodes around a controller with a star topology, or being mesh connected with thousands of nodes acting together. A recent example of very large system is the car parking network deployed at San Francisco with more than 4000 nodes [Swedberg 2007]. Another dimension of WSN is mobility. Some networks have fixed sensing positions with mobiles crossing these positions, others are *fleets* having complete mobility. A case (curious) example is a *Smart Farm*, where the cattle is equipped with wireless sensors and GPS[Wark et al. 2007].

The main stream in sensor networking is the 802.15.4 open architecture also known as ZigBee [Aggelou 2008]. ZigBee provides the possibility to connect ten of thousand of *static* sensors, providing routing and medium range connections. Usually nodes receive a small operating system such as TinyOS [Matischek 2008], and they are programmed using a C derivative such as NesC[Gay et al. 2003].

As the acceptance of WSN requires faster developments done by application experts, it is now necessary to provide more efficient techniques to program, at least, the simplest applications. WASP [Bai et al. 2009] is a step forward in this direction with a claim that applications could be developed in very short cycles, far better than existing WSN dedicated languages. WASP is using NesC as a back-end and thus, has the restrictions of NesC/TinyOS.

Instead of using low level development approaches, we are developing a two-step top-down flow allowing to build optimized *specific systems*. The idea is to follow architecture development methodologies, by concentrating first on distributed system problems, and, second, by synthesizing optimized code for node hardware.

The main motivations to build an abstract representation of target networks are as follow:

- WSN applications can be very different in nature, with *difficult* distributed problems to solve. An example is the necessity to route security information in the presence of failures as it would be the case for a railway sensing system deployed linearly on kilometers. Another example is

the unpredictable topology of a *mobile fleet* where each node must send data or alerts to a moving gateway.

Basic problems in these situations are dedicated or adaptive routing algorithms, data collection control, self healing in the connectivity.

- node architectures result from a careful study of application criteria. This is especially a problem for the choice of capacity of the memory used for local data and message buffering, and for the choice of communication periodicity.

  Furthermore, if emerging *mixed-signal* technologies are to be used, the programming tools must use modules merging software tasks, on chip routing, and configuration for interface blocks. C derivatives will not work in this situation, but object oriented or process oriented design tools can still operate[Doboli and Currie 2007].

As wireless communications imply emitting and receiving according to a known schedule, the sensor network has a synchronous behaviour. Details on how the actual network will work are thus replaced by the abstraction of distributed *synchronous communication model.*

This is a well known abstraction for distributed processes exchanging messages periodically with their direct neighbours. The system achieves *locked step* progresses with the advantage to simplify design and proofs of algorithms due to the quasi-synchronous temporal property. Numerous algorithms and systematic approaches are known to work in this context [Lynch 1998]: mutual exclusion, leader elections, spanning trees, expression reductions, transactions, failure recovery, ... Programs for the synchronous model can be expressed as loops over sequence of procedure calls corresponding to distributed computations that nodes agree to do.

To reach the simulation level, a design flow would start with the specification of the sensor distribution at geometric level, from which a communication connectivity can be automatically deduced. This provides a distributed process system organization, on which we can add local sensor behaviours. At this point, we have an executable specification suitable for simulation of a future real deployment. Program development is mostly achieved during this first stage.

In a second stage, the local sensor behaviours are translated into code for the targeted node micro-controllers, or virtual machines supported by the nodes. This is a technical activity that can rely on compilers, libraries for local data collection, for control, and on available distributed services.

The whole framework has models and tools for specification of a planar geographic distribution, for process system connectivity, and for synchronous communication activity. As we have a highly concurrent communicating system, it is convenient to refer to an existing concurrent programming model. The CSP programming style allows us to represent sensor nodes as processes, with channel communications implemented by wireless communications. Practically, the process system is an Occam program suitable for massive parallel simulation. Each process receives array of channels as parameters, and executes a loop operating communication, and local processing. Models and tools are developed in Smalltalk-80.

The paper describes the domain requirements and the internal graph model. This model can be addressed in ways depending on the application. Section 2 presents random and textual input of networks organizations. In section 3 we describe how Occam programs are produced according to the synchronous communication model. A sample of these programs is provided in appendix B. Section 4 presents an algorithm conforming to the synchronous model, this algorithm allowing to decide communication time slots for this model (time division MAC access).

2. Requirements and solutions

2.1 Connectivity

The simplest situation is the case of sensors deployed in a particular location: a house, a plant, a greenhouse. For largest scale, WSN are experimented on towns, forests, roads, railways, etc...

Sensors can also be moving if they are carried by animals, cars, boats, ...

For sake of simplicity we will ignore the dynamic evolution of the network, accepting the idea that we will be able to manage these evolutions as a succession of phases where the node interactions are static, or quasi-static, meaning static with failures. The dynamic evolution will be studied as a succession of quasi static phases.

Given a distribution of sensors, we consider that the connectivity is given by the capability of one node to send and receive to neighbour nodes. Again, there is a large amount of variability there, since some nodes can tune amplification to extend or reduce the neighborhood. We suppose that sensors emit in a zone represented by a circle, having a given radius, in which other sensors can listen communications. This is known to be a coarse approximation.

The connectivity is defined by a graph which vertexes are sensors and edges are links that can carry communications from one node to another node.

2.2 From geometry to connectivity

Deploying a WSN means planning sensor positions. The first idea of field engineers will probably be to get a map and point particular positions, with the purpose to devise how sensors can connect together producing a real network.

In other situations, one can rely on random dispatching of devices to cover a particular surface, or to observe mobiles moving according to social or natural laws.

The key points are the needs to represent:

1. geometric distributions, in one or several dimensions
2. a network abstraction represented by a graph

3. communication capabilities to predict graph from distribution

Geometric distributions are abstract, or real. In large scales the exact geographic position can be very important to practically locate the sensor. In other cases, distances between position are enough to make decisions. Finally, the engineer will be interested to get a tool showing a view on the zone to be controlled, a scale meter, and to let tools decide what is the topology obtained by a sensor distribution.

Geometric distributions can be modelled in a small class hierarchy allowing to describe these particular situations, and conversions from one to another. As an example, a drawing on a view needs to be translated in a set of GPS positions to allow sensor setups. Reciprocally, given a collection of GPS positions collected by a WSN, we will need to draw it on a computer screen to represent mobile positions or their moves.

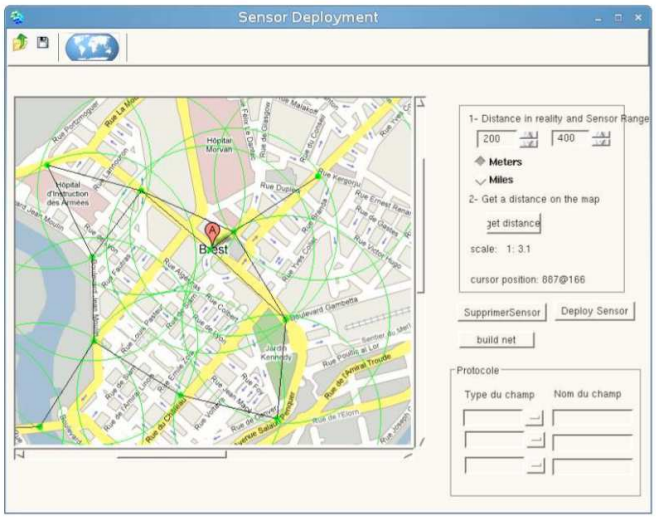

Figure 1. From distribution to connectivity: node distribution is decided to obtain wireless connections. Circles represent area where other sensors can (likely) receive the center messages.

2.3 Models and algorithms

Basically a distribution of sensors is simply a collection of sensors. Each of the sensor has a location expressed in a class according to the current space reference.

The network structure relies on several classes for nodes, abstract graph and topologies. Nodes have collections for input and output neighbours and support for naming the program to be executed. A graph is a set of nodes, and a topology provide more context information such as symbols used for messages, and provision for textual expression of networks[1]. The listing below is the textual representation of a random distribution which process level organization is shown figure 2.

[1] These classes are also suitable to describe lot of other situations, in particular circuit architecture organizations.

Listing 1. Network specification

```
genRange640Points20

messages none defined.
P2 { P7, P14 } Node
P3 { P5, P15 } Node
P4 { P17, P20 } Node
P5 { P3, P15 } Node
P6 { P11, P13, P18 } Node
P7 { P2, P8, P19 } Node
P8 { P7, P19 } Node
P9 { P10, P12, P19, P20 } Node
P10 { P9, P12, P17 } Node
P11 { P6, P13, P18 } Node
P12 { P9, P10, P19 } Node
P13 { P6, P11, P18 } Node
P14 { P2 } Node
P15 { P3, P5 } Node
P17 { P4, P10, P20 } Node
P18 { P6, P11, P13 } Node
P19 { P7, P8, P9, P12 } Node
P20 { P4, P9, P17 } Node
```

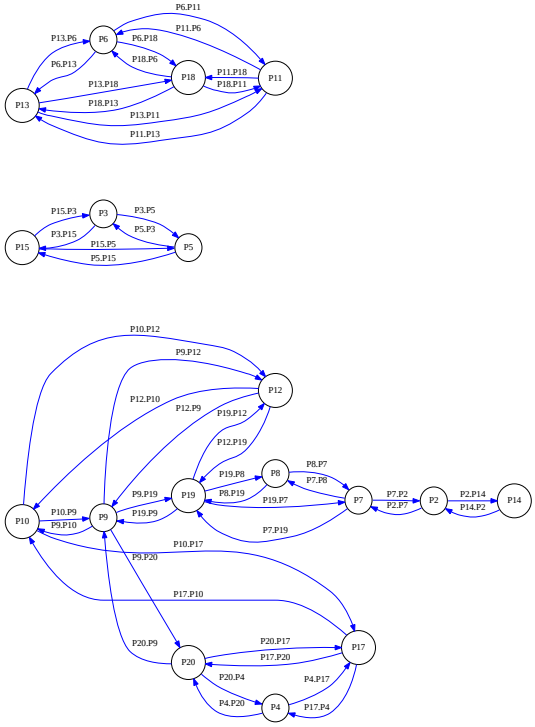

Figure 2. Process organization for a 20 nodes random system. Geometric space is a drawing view of 640pts, sensors can reach neighbours at the same distance. Some nodes are isolated

Building the network from the geometric distribution is the following loop (simplified):

Listing 2. From distribution to system

```
graph := Dictionary new.
distribution do: [ :eachNd |
  neighbours := Set new.
  distribution do: [ :otherNd |
    (eachNd canReach: otherNd)
      ifTrue: [ neighbours add: otherNd ]].
  graph add: (eachNd -> neighbours)].
^graph
```

The two models for distribution and networks are interesting for the expression of organizations in several points of view:

- topologies can be specified for different situations, the simplest ones being random distribution, or manual positioning as shown in this section.

  Further possibilities are numerous, including the control of distributions from simulator moving sensors according to given laws: car traffic, walk in a mountain, tide in a bay, etc...
- topologies can translate into graphs representing sensor networks, this level being reusable for lot of other abstractions.
- these graphs are denoted by a syntax shown above, where each node as an associated program or procedure (or method). Graphs can be translated into process logic organizations as shown figure 2. This is obtained by rewriting into the *dot* language for *Graphviz software.*

The next section will show that a major application is rewriting to an executable concurrent program.

3. Representing WSN as concurrent programs

In the first stage of developments, we must concentrate on the fundamental questions and their possible solutions. The domain is a set of small systems, possibly in unknown number, communicating together, possibly in a changing network organization.

Our beacons are the knowledge developed in distributed algorithms, the state of the art in sensor networking (open systems), and mobile routing techniques explored in Mobile Ad-hoc Network (MANET)[Macker and Chackeres 2009]. We can also rely on system specification mechanisms proposed by the Concurrent Sequential Processes (CSP) paradigm, and its associated programming languages and compilers Occam, and Occam-Pi for mobile systems[Inmos b] [Welch and Barnes 2004].

The introduction of the domain is the fact of Mark Weiser's vision[Weiser and Brown 1996] of *ubiquitous computing* where lot of communicating small devices would help people non intrusively. Today this vision is a reality thanks to communication and integration technologies.

We need intermediate level representations of the real sensor deployments with the possibility to develop and check solutions. The situation is similar to the creation of a virtual machine, except that the real code will be distributed on distant elements. A system language can help.

3.1 WSN modelling using Occam

Occam is attractive because of its complete set of primitives for the expression of concurrency:

Par constructs for the development of structural parallelism controlled by barriers,

Channels as the single primitive for synchronization and communication

Alternatives specifying non-determinist behaviours

Protocols to model sequence of exchanges on channels

Timers allowing to introduce clocks, delays and time-outs into programs

Occam has several compilers that generate native i386 processor code, or intermediate code for virtual machines (VM). A threaded run-time allows to simulate hundreds of processes on shared memory multi-processors [Ritson et al. 2009], and there is documentation and free software allowing to build or evolve VMs.

Occam and its tools appear as a serious support for modelling and testing WSN behaviours. As large irregular concurrent programs are difficult, if not impossible, to produce by hand, tools described section 2 will considerably ease the generation of complex systems from WSN behaviours.

3.2 Occam program skeleton generation

An Occam program is a structural description of a concurrent activity. If the language, and associated instruction set, have relatively high level system primitives, it remains that the process organization, and the communication channels must be given in the program, and is checked effectively by the compiler. In the case of sensors, we have irregular system architectures.

By analyzing the network model from section 2, we can produce everything necessary for a translation to Occam:

- channel declarations (perhaps thousands!)
- grouping of channels in aliases used as process parameters (perhaps hundreds)
- computing process fan-in and fan-out, making buffer space reservations
- starting the WSN as a set of cooperating processes
- declaring procedure skeletons for process found in the network
- tracing.

The expression of the network as a process system is a concurrent program showing process, procedures, channels, that compiles and executes on empty procedure code.

The next step is to add an execution model suitable for WSN. A credible choice is the synchronous communication model, that will fix local sensor behaviours into automata looping on communications, state evolution, and next message productions.

3.3 Synchronous communication model

This model is known in the domain of distributed algorithms as a good starting point for algorithm design and proof. Each node contributing to a computation must execute repetitively a loop including a communication phases, and a computation phases. During the communication phases, each node must send messages to his neighbours, and accept messages from his neighbours.

In this way, we produce a kind of global clock that mimics a computer clock, and we can more easily reason on the system behaviour.

To introduce this model in our Occam programs, we simply need to produce an infinite loop denoting communications with neighbours. The following code illustrates the local program behaviour.

Listing 3. Synchronous model

```
-- Procedure definitions
PROC P( []CHAN OF proto in ,
[]CHAN OF proto out, -- folded
VAL INT  identity) -- folded
   -- messages declaration P
  [4]BYTE inMessages:
  [4]BYTE outMessages:
   -- Code of procedure P
  SEQ
    -- initialize
    SEQ tours = 0 FOR 10
      SEQ
        PAR
          PAR i=0 FOR SIZE in
            in[i] ? inMessages[i]
          PAR j=0 FOR SIZE out
            out[j] ! outMessages[j]
        -- local procedure call here
:
```

All process will thus execute this template. We can notice the declaration of channel groups as parameters, and computed maximum dimension (4) for incoming and out-coming message arrays. Here, the loop count has been fixed at the limit of 10 synchronous steps by the generator. The communication is a parallel construct to avoid deadlock during execution. There are placeholders for initialization and state transition code implementing the distributed behaviour.

It is also noticeable that each process receives an identity as an integer. Using this integer, it is possible to access a global array, and configure the process from values produced by the generator. The appendix B provides more details with a sample of program structure.

3.4 Adding behaviour

Our flow has reached a stage where a complete process system skeleton can be produced automatically from a variety of WSN deployment situations.

This system will accept to loop on empty transformations: messages do not carry anything, and incoming information is even ignored by our process. We have a representation of a distributed computer of arbitrary organization, executing repetitively a NOP instruction.

At this stage, we need to add a local behaviour. Following propositions from Nancy Lynch model in [Lynch 1998], we propose to fix the behaviour in a procedure called after the communication construct. This procedure will process parameters for:

current incoming message buffer array

next out-coming message buffer array

state a record grouping local variables

The evolution procedure has the responsibility to analyze the new messages making decision on local variable evolutions. Then it produce next phase messages into buffers.

Such procedure can be produced *manually*, by synthesizing automaton programs described in a given syntax, or they can follow a template for encoding/decoding network protocols.

The synchronous model techniques are convenient to design and check distributed algorithms. By following a regular pattern, they open the way to reuse of validated library code, possibly manipulated by higher level languages.

3.5 A routing protocol case study

Our hypotheses is that some sensors need to send data to a known particular destination. This can be a device with more memory support, or a gateway providing WSN interface to Internet, as example.

We must consider two situations:

1. a network where all nodes are in range.
2. a mesh connected network

In the first case, routing is useless since sending data from one point to another is direct. In the second situation, availability of a routing protocol is necessary, as example, an algorithm based on table exchanges between cooperating nodes.

This protocol appears as a particular procedure in charge of routing table maintenance. This procedure is called upon receiving a table from a neighbour. Its role is to compute

nodes	user	real	channels
10	5.8s	1.1s	14
100	7.4m	1.3m	1510
400	149m	42m	24496

Table 1. Simulation time for 100000 empty synchronous loops. Quad Xeon multiprocessor providing 8 processors. Observe that the execution time relates to the number of channels (graph edges, also meaning communication operations), and not the number of processes.

the shortest path to destinations in view of further routing operations.

As the network can be mobile or include mobiles, the entire routing tables computation must happen regularly.

The algorithm is divided in two parts. In a first stage, each node transmits its own table to its neighbors. In a second stage, nodes update their table by adding missing nodes and compute a minimum hop number toward each node. The table update procedure is given in appendix C to illustrate how behaviours are described. Validation is obtained by executing a simulation with routing interleaved with random data propagation.

3.6 A summary on system simulation

At simulation level, WSN are represented as communicating processes with arbitrary organizations. We have presented a method for applying abstract algorithms, and producing simulations. These simulations can be executed on concurrent machines supporting a thread level distributed Occam kernel (table 1).

Another claim is the capability to produce tools for distributed algorithm development by following the synchronous model methodology. WSN behaviours can now be expressed as sequences, loops, conditional calling procedures. The programmer view is the development of algorithms operating on a set of processes, with total abstraction of communications.

Now, we need to explain how the simulation level (similar to VHDL) could be used to produce real processor code and real communications (similar to real circuit layout). This is discussed in the following section.

4. Synthesis for real systems

Sensors is a maturing technology. Recent products include circuits with the following characteristics:

- 2.4Ghz transceiver with SPI serial bus for control and data exchange. Spread Spectrum, low power, 16 byte buffers, address decoding and optional acknowledgement of transactions.
- Reconfigurable logic/analogue circuit with 8bit microcontroller, small RAM and flash memory, general purpose I/Os, SPI interface.

Because the controller can directly integrate analogue signals, a couple of these circuits is enough to build a wireless sensor.

Other options include ARM system on chip, GPS receiver on one chip etc.. ARM can be programmed using free GNU tool chain, which is not the case for the first controller. In each case, systems have very low memory capacity, in particular for data and stack management.

4.1 Options for implementation

A reference for low level implementation is simply the target processor for Occam. This processor (transputer) existed several years ago, and still survives due to its instruction set (TIS).

The Kent Retargetable Occam Compiler (KROC) proceeds in two passes producing an intermediate code suitable for Occam, then translating to i386 native code. TIS has process level semantics with instructions for launching processes, communicating on channels, computing barriers, managing timers, etc... TIS is documented by [Inmos a], has a simulator VM written in C, and has a portable interpreter initiative called the *Transterpreter* [Jacobsen and Jadud 2004].

Writing such a VM is not very difficult due to the abundant documentation and examples available. As there is a compiler available, a first option to cover WSN needs is to implement a VM adapted to the sensor framework removing useless instructions, and adding new ones to support local control directed to sensing devices, and the synchronous communication model. TIS code is very compact and perfectly suitable for spreading programs on WSN, securely and at low cost. This can be a decisive advantage for the VM approach.

Another option for practical support is to build a synthesizer for C, if there is some chance to compile C to a controller, or native processor code in the other case.

The practical communication issue is discussed with a proposition in the following section.

4.2 A link access model

Wireless transmissions are inherently multiple access on a shared medium. Basic communications are always broadcast, and point to point messaging is a consensus on a channel choice and how to decide to receive a packet.

Collisions can happen, and this is more likely to arise in some well known situations. The exposed terminal and the hidden terminal problem are two of them. Both are well explained in the literature [Karl and Willig].

Two nodes sending data at the same time and at the same frequency to a third node is an example of a potential collision situation. To reduce packet losses, the two nodes

can send their data on different time slots, or using different frequencies.

Knowing that, we can consider to use both time and space scheduled transmissions. Here, space is the sensor distribution space, as discussed in section 2.

Each synchronous cycle will now be divided into a silent period and a communication period called a *superframe*. Each superframe is split into *time slots*.

We can use the terms of Time Division Multiple Access (TDMA) for time access, Frequency Division Multiple Access (FDMA) being for frequency allocation. Spread Spectrum technique provide channels on a shared frequency band.

TDMA is a communication model that provides each node with a non ambiguous time slot for emitting. By non ambiguous we mean, that two nodes having a same other node in their neighborhood should not transmit at the same time. In a similar way, FDMA will need to allocate one ore several frequency band for each node in the system.

The problem is similar to solving a graph node coloring problem where the graph is the wireless network as discussed section 2. The graph coloring problem is solved by assigning two different colors to two nodes linked in the network. In our cases colors represent a time slot in the TDMA context or a frequency in the FDMA context (see figure 3)

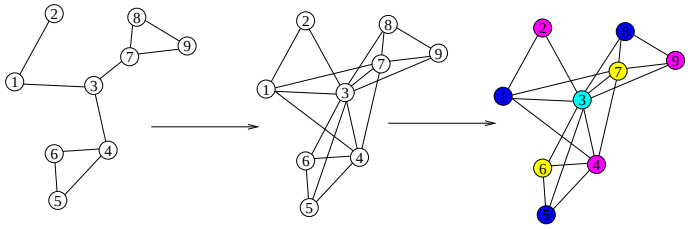

Figure 3. The figure shows different networks used during the coloring phase. First scheme is the wireless network, two nodes are linked if they are at range. Second one is the augmented graph, i.e. the first one with edge linking two nodes having the same neighbor. Last network is the result of the graph node coloring algorithm

The algorithm proposed for distributed allocation of communication channels is as follow:

1. an appearing node randomly chooses a slot number (color), and broadcast a slot request to its neighbours. The request is propagated at a distance of 2 around the sender.

   A special slot is used to arrange color allocation.

2. If one of the node crossed by the request uses the chosen color, an alternative color is proposed to the requester.

   By bounding the number of nodes crossed by the request by 2, only neighbors at a distance of one or two hops will be hit by the request. Therefore we can set an upper bound concerning the reception of the answer to 4 synchronous cycles.

3. The color is validated if the asking node doesn't receive any request four cycles after emitting a request.

Few observations can be made relative to sensor distribution and relative to the synchronous model applicability.

Sensor generally do not need to speak all the time, but periodically, or on-demand. One can see the general behaviour of a sensor as a cycle, including power saving mode, then the sensor awakes, process data and enter a communication activity. For this to work, it is necessary to maintain a shared clock reference.

During communication period, each sensor must speak and listen. In a large network, there will be shared slots due to coloring. In small, grouped networks, there can be a complete connectivity inside the network. In this situation, it is necessary to have one slot for each node. Another interesting question is the relation between the synchronous communication model, and TDMA.

As we have a turn allocated for each sensor, we can implement the neighbour communication phase by sending our messages at this turn, listening the rest of the superframe, or only one part of the slots if the neighbour slots are known.

5. Conclusion

We have described a top down approach for WSN programming. An associated framework is being built centered on a graph model. This model can be addressed in different ways such as texts, interactive maps[Kremer and Osmont 2009], or random distributions. It can be translated in different ways, and an example are Occam programs which processes execute loops embedding communications and local processing according to the synchronous communication model. The aim of the framework is to focus on distributed algorithms design, use and validation, prior to sensor code generation. Concurrent simulation of intermediate programs can be obtained using the KROC compiler targeting multicore processors. As it is, the framework has been used to simulate network activities such as route management, effective routing (RIP, AODV, DSDV), naming. Very large random networks have been produced to check correctness of the solutions.

It is a known question that the time-driven simulation achieved on the system process can be sub-optimal compared to an event driven simulation that can reduce the number of channel synchronizations. An improvement will thus to produce a different execution sequenced by an appropriate scheduler supporting a virtual time.

It is also known that more support could be obtained from Occam variant (Occam-Pi) that allows to describe mobile channels, and mobile barriers, but it appeared risky to use this support, due to the necessity to control future sensor mo-

bility by outside considerations managed in the Smalltalk-80 framework.

As shown by the current Occam code generator and dot file generator the framework can easily be extended to address a varietu of situations, including the need for simulation using Smalltalk-80 process networks[Goldberg and Robson 1983].

The second stage for real code synthesis is under investigation with the creation of a virtual machine for sensors. The current direction is to execute a subset of the *Transputer Instruction Set* (TIS), replacing channel operations by call to the local wireless communication library.

A. Generation flow and data structures

Listing 4. Sample textual specification

```
<methods>
<class-id>
AlgoDis.NetworkTopologies</class-id>
<category>compiling</category>

graph1
"sample network"

messages M1 , M2 .
P1 { P2, P3, P4} Pim
P2 { P1, P3 } Pam
P3 { P1, P2 } Poum
P4 { P1, P2 ,P3 } Poum
</methods>
```

A.1 Textual specification

A parser `NetworkTopologies` translates a textual specification into an equivalent object representation.

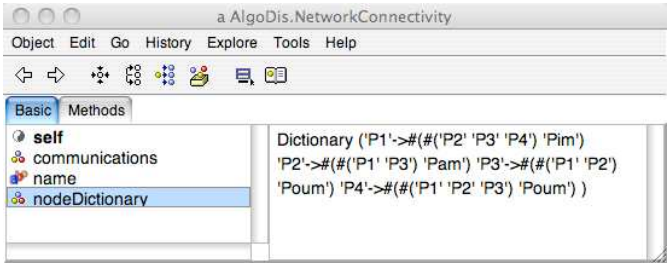

Figure 4. Representation of the network description after parsing is a dictionary which keys are node names and values are couple arrays for fan-out and process binding.

A.2 Graph model

The sample program has been develooped as a model, figure 6 shows the data organization of the resulting graph. The test method `graphSample` demonstrates the simplicity of the specification and its development.

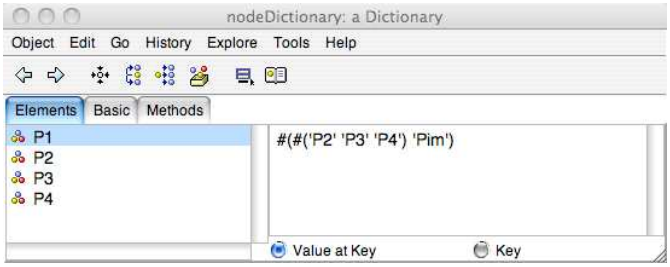

Figure 5. The first element in a couple is the set of reachable nodes, the second element is the name of a procedure to be executed.

Listing 5. Test method

```
<st-source>
<methods>
<class-id>
AlgoDis.NetworkTopologies class</class-id>
<category>testing</category>

graphSample

 "build a synchronous Occam program"
 | net graph text file |
 net := self new graph1. "parser model"
 graph := net buildGraph. "graph model"
 text := graph programMain. "Occam text"
 file := 'syncModel.occ' asFilename writeStream
 file nextPutAll: text; close.
</methods>
```

B. Synchronous program sample

The following listing shows the Occam skeleton developed for a 4 node ring.

Listing 6. Ring executing an empty loop

```
--generated at June 23, 2009 10:17:43 am
--generic synchronous network model
--Bernard Pottier. UBO
--ring5
--
--messages null search leader.
--P0 { P1 } Node
--P1 { P2 } Node
--P2 { P3 } Node
--P3 { P0 } Node

VAL [4][2]BYTE NetProcess IS [
  "P0", -- id: 1
  "P1", -- id: 2
```

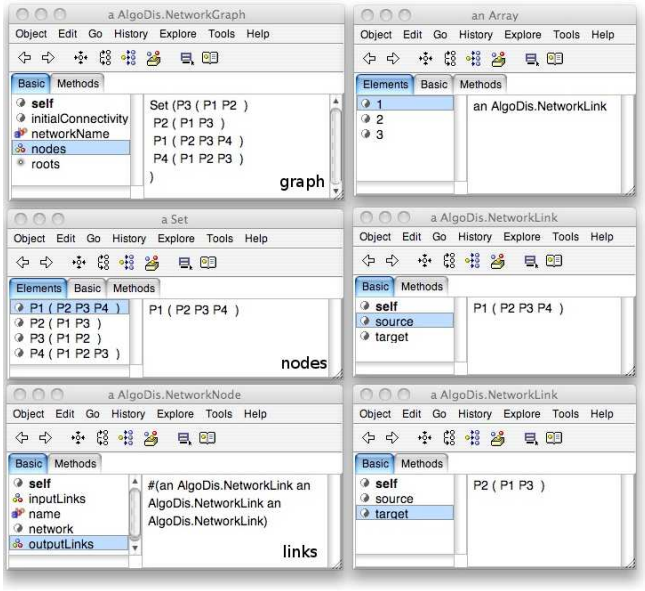

Figure 6. First column shows the graph including a set of nodes, each node having two collections for input and output links. On the right, the first output links has source and target NetworkNodes.

```
  "P2", -- id: 3
  "P3"]:

VAL [4][4]BYTE NetProcedure IS [
  "Node", -- id: 1
  "Node", -- id: 2
  "Node", -- id: 3
  "Node"]:



PROTOCOL ring5.proto IS  BYTE:


-- Procedure definitions
PROC Node( []CHAN OF ring5.proto in ,
           []CHAN OF ring5.proto out,
           VAL INT  identity)
  -- messages declaration Node
  [1]BYTE inMessages:
  [1]BYTE outMessages:
  [2]BYTE MyName:
  -- Code of procedure Node
  SEQ
    SEQ index = 0 FOR SIZE MyName
      MyName[index] := -- folded
          NetProcess[identity][index]
    SEQ tours = 0 FOR 100000
      SEQ
        PAR
          PAR i=0 FOR SIZE in
            in[i] ? inMessages[i]
          PAR j=0 FOR SIZE out
            out[j] ! outMessages[j]
        -- dummy behaviour
        SEQ i=0 FOR SIZE in
          outMessages[i]:=inMessages[i]
:

PROC ring5()


  -- Channel declarations
  CHAN OF ring5.proto P0.P1:
  CHAN OF ring5.proto P1.P2:
  CHAN OF ring5.proto P2.P3:
  CHAN OF ring5.proto P3.P0:

  -- Channel table declaration for nodes
  P0.out IS [ P0.P1 ]:
  P0.in IS [ P3.P0 ]:
  P1.out IS [ P1.P2 ]:
  P1.in IS [ P0.P1 ]:
  P2.out IS [ P2.P3 ]:
  P2.in IS [ P1.P2 ]:
  P3.out IS [ P3.P0 ]:
  P3.in IS [ P2.P3 ]:


  -- Program Body
  PAR
    Node(P0.in, P0.out,1)
    Node(P1.in, P1.out,2)
    Node(P2.in, P2.out,3)
    Node(P3.in, P3.out,4)
    -- End of program body
:
```

C. Code for route management

Listing 7. Route maintenance procedure

```
PROC UpdateTableRoute(TableRoute tLoc, TableRoute tRecue, VAL INT canal)
  INT trouve :
  Route r :

  SEQ

    SEQ i = 0 FOR tRecue[dernier]
      SEQ
        trouve := 0
        SEQ j = 0 FOR tLoc[dernier]
          IF
            tLoc[tabR][j][network] = tRecue[tabR][i][network]
              SEQ
                trouve := 1
                IF
                  tRecue[tabR][i][distance] < tLoc[tabR][j][distance]
                    SEQ
                      r[network] := tRecue[tabR][i][network]
                      r[canal] := canal
                      r[distance] := (tRecue[tabR][i][distance]) + 1
                      r[qual] := tRecue[tabR][i][qual]
                      r[nextRouteur] := tRecue[tabR][i][network]
                      getRouterId(tLoc, canal, r[nextRouteur])
                      ajouterRoute(tLoc[tabR], j, r)
                  TRUE
                    SKIP
            TRUE
              SKIP
        IF
          trouve = 0
            SEQ
              r[network] := tRecue[tabR][i][network]
              r[canal] := canal
              r[distance] := (tRecue[tabR][i][distance]) + 1
              r[qual] := tRecue[tabR][i][qual]
              r[nextRouteur] := tRecue[tabR][i][network]
              getRouterId(tLoc, canal, r[nextRouteur])
              ajouterRoute(tLoc[tabR], tLoc[dernier], r)
              tLoc[dernier] := (tLoc[dernier]) + 1
          TRUE
            SKIP
:
```

References

George Aggelou. *Wireless Mesh Networking, with ZigBee*. Mc-Graw Hill Communications, 2008.

Lan S. Bai, Rober P. Dick, and Peter A. Dinda. Archetype-based design: Sensor network programming for application experts, not just programming experts. *IPSN'09, San Francisco*, April 2009.

Alex N. Doboli and Edward H. Currie. *Introduction to Mixed-Signal, Embedded Design*. Cypress semi-conductor, 2007.

David Gay, Philip Levis, David Culler, and Eric Brewer. NESC 1.1 language reference manual. May 2003. URL `http://nescc.sourceforge.net/papers/nesc-ref.pdf`.

Adele Goldberg and David Robson. *Smalltalk-80, the language and its implementation*. Addison-Wesley, 1983.

Inmos. *Transputer instruction set, a compiler writer's guide*. Inmos, a.

Inmos. *Occam2 reference manual*. Prentice-Hall International, b.

Christian L. Jacobsen and Matthew C. Jadud. *Communicating Process Architecture*, chapter The Transterpreter, a Transputer Interpreter, pages 99–106. IOS Press, 2004.

Holger Karl and Andreas Willig. *Protocols and architectures for wireless sensor networks*. Wiley.

Guillaume Kremer and Jimmy Osmont. Reseaux mobiles de capteurs: Outils d'étude et de simulation. Master's thesis, Master informatique – Universite de Brest, May 2009.

Nancy Lynch. *9.49-333.7Di(P)-3.04959(l)-g.81672(0)2.16739(o)4.81672(r)02.167K9Di(n-3.049597dH51.16739 2008.81668(B)-0.70 1249458(A)-0.81672(M)2.60495*
Cla.883564(i)-2.16739(r)-1.-8804(,9-S247.33(S)-3.04982(m)-0.699782(c)4.6.793(n)4.81603(b)4.81672(e)2.3498995-
http:///tfosfjsour/nefs/neep/rse/s0.882202]TJ /R9 8.96.0142f 225.239 0 Td [(.)-4.28325]TJ -312.72 -13.91332.242(Sa
.rne.eteep938 T tir35.123.ppt095(o)t r tel http://tflcflfs . 8 ete(9)H51.916(G)-0.704959((u)-1.584(g)-4.816