# Lustre as a System Modeling Language: Lussensor, a Case-Study with Sensor Networks

Florence Maraninchi[1], Ludovic Samper[2], Kevin Baradon[3], and Antoine Vasseur[3]

1. VERIMAG, 2, av. de Vignate, F38610 Gières.
2. France Télécom R&D and VERIMAG;
3. INPGrenoble/Telecom and VERIMAG

**Abstract.** We describe how we use Lustre to build global and accurate executable models of energy consumption in sensor networks, intended to be used for both simulations and formal validation. One of the key ideas is to build a component-based global model, in such a way that various abstractions of the same model can be derived by unplugging a component and plugging a more abstract (or more detailed) one. This ability to play with various abstractions that can be formally compared with one another is essential for a virtual prototyping approach connected to formal validation tools. We comment on the properties of Lustre and its development environment that make this approach feasible.

## 1 Introduction

### 1.1 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are distributed computer systems composed of a large number of small sensor nodes. There are many potential application areas, ranging from defense and surveillance to health or intelligent homes [4]. The nodes have three main tasks: sensing their environment, processing the data and communicating with the other nodes. All the nodes are identical, except one or several *sink* nodes. A sink monitors the network. It collects the data and sends requests to the sensor nodes. Nodes do not have enough power to reach the sink directly with their radios, therefore communications are performed in a multi-hop way. A wireless sensor network has no infrastructure, i.e., nodes do not have any a priori knowledge about the rest of the network. Thus a Wireless Sensor Network must be able to self-organize. Nodes cooperate for this self-organization and also along the whole life of the network to achieve the requested service. Finally, routing protocols have to be tailored for WSNs (see, for instance [3]), in order to take new constraints into account, among which: in WSNs, data flow from a particular region to the sink(s) whereas, in traditional networks, any node may need to communicate and establish a route with any other; it is usually not possible in WSNs to rely on a global addressing mechanism; in the vicinity of a phenomenon, several nodes will sense the same values and thus the same data may be generated; etc.

### 1.2 Virtual Prototypes of Sensor Networks

A sensor network may be considered as a whole, as a new kind of computer system dedicated to one particular application. It is an *embedded* system, reacting to the stimuli of some physical environment. It is also subject to the usual constraints of embedded system design: resources are scarce, and it is very difficult, if not impossible, to modify a sensor network's behavior once it has been deployed. Moreover, the sensors are usually powered by a battery that cannot be recharged. They should therefore have the lowest consumption possible to maximize the network lifetime.

One of the main challenges is to perform energy-aware design. The problem is difficult because all the elements of a sensor network have an influence on energy consumption: the hardware of a node, the sensors, the medium-access-control and routing protocols, the application itself, the initial self-organization phase, and even the physical environment that stimulates the sensors (see, for instance [24], where we showed that a precise modeling of the physical environment is compulsory for a realistic estimation of the energy consumption).

The design of an energy-"optimal" solution is probably out of reach because of all the interacting criteria, and the complexity of some of the elements that have to be taken into account. If there is no way to compute an optimal solution, then the only solution is to build complete solutions and then to evaluate them. Moreover, since a sensor network includes dedicated hardware, it may be long and costly to build a complete solution before evaluating it.

For all these reasons, the usual approach is to build a *virtual prototype* of a sensor network, and then to perform simulations or mathematical analyzes in order to evaluate the energy consumption. This is the approach taken by people who design new protocols, and show their benefits using a network simulator. In all these approaches, a lot of *abstractions* are necessary, in order to build manageable models of very large systems (thousands of nodes). For instance, the energy consumption may be evaluated by counting packets, and associating a worst-case estimated energy with the transmission of one individual packet. In the section "related work" below, we review the main approaches for the virtual prototyping of sensor networks.

### 1.3 Contribution of the paper

In this paper, we describe our experiments in using a synchronous language, namely Lustre, to build a global and executable model of a sensor network, including all the elements that influence energy consumption: the details about the hardware of a node, the code of the protocols and the application, an executable model of the physical environment that stimulates the sensors, and also a model of the physical medium in which the radio communication occurs. Lustre is an appropriate language for building such a detailed model, especially when it comes to describing the detailed energy consumption of the hardware and its relationship with time.

The second very important point is that Lustre allows to build a clean component-based model. This is crucial because complete models of sensor networks are huge, and it is always necessary to abstract them, even for simulation purposes only. If a global model is clearly structured into well-defined components, it means

that one can hope to replace one component by a more abstract one, and get a new global model, more abstract then the original one. We show that LUSTRE provides such a modular-abstraction framework.

Finally, LUSTRE is connected to various validation tools, ranging from automatic test case generation to formal verification by means of model-checking or abstract interpretation techniques. This means the model we build can be directly given as input to these tools.

The LUSTRE model is 1500 lines long. It has been developed by K. Baradon and A. Vasseur, two master students of the Telecom department of INPGrenoble. It took approximately 7 weeks, including tests and documentation. This has been made possible by a first experiment in writing global models of sensor networks, that was conducted using the language REACTIVEML [19, 24]. GLONEMO (for GLObal NEtwork MOdel) is an executable model of a sensor network written entirely in REACTIVEML, and it is quite efficient. It is itself inspired from a first use of REACTIVEML for modeling networks [20].

The paper is organized as follow: section 2 lists the elements that have to be taken into account when building an accurate global model of a sensor network; section 3 presents the main structure of the LUSTRE model; section 4 details the components of the model and the way they are coordinated; section 5 presents existing tools and methods designed to study sensor networks; section 6 lists the current uses of our model, and section 7 concludes.

## 2 Aspects to be Taken into Account in a Realistic Model

### 2.1 Consumption of the radio and Behavior of the MAC protocol

The radio is the part of the node that consumes most, and mainly in emitting mode. It is clear that the radio should not function at maximum power all the time. In sensor networks, the Medium Access Control (MAC) protocol layer (the one that monitors the radio) is designed in such a way that the radio spends a lot of time in some idle mode, and very short periods in emitting mode.

In an accurate sensor network model, the various *modes* of the radio and their associated consumption should be detailed. Moreover, the way the MAC protocol triggers mode changes has to be described. This is because we want to observe properties related to energy consumption. Other properties like latency, throughput, bandwidth utilization or fairness are secondary.

### 2.2 The CPU and the Memory

Even if the radio consumes a lot, the energy used to process data cannot be neglected. According to Yuan and Qu [28], the processor is responsible for a consumption of 30 percent of the total consumption of the node. Moreover for some MAC protocols the micro-controller can be responsible for more than 90 % of the total energy needed to receive one data packet [21]. A technique used to optimize the energy consumed by the micro-controller is *Dynamic Voltage Scaling* (DVS). DVS consists in adapting

dynamically the voltage of the micro-controller according to the load. This modifies the tradeoff between the consumption and the efficiency of the MCU. When the voltage is low, the consumption is low too but the micro-controller works slowly. This idea can be used in sensor networks [28].

If we want to reflect such technical solutions in our global models, this means that we should also detail the running modes of the CPU (a small number of discrete voltages is enough, the DVS is not driven in a continuous way). We should also describe how the mode changes are triggered, and by whom.

The CPU DVS may be driven by explicit operations in the object code of an application program, if a static analysis has identified pieces of the program where the load is low. If there is no such sophisticated analysis available, the CPU DVS is usually controlled by the operating system. In sensor networks, it may also be the case that the CPU is awaken by some activity on the radio (when there will be some data to process). In the sequel, we consider commands from the application. Commands from the radio could be modeled with the same technique.

The consumption of the memory is less important but researches are conducted on this topic [6, 16]. If we want to take memory consumption into account, we should include the description of the memory consumption, depending on the type (RAM, Flash, ...). Some memories can have a `standby` mode in which they cannot be read or written to, but consume less. In order to read or write, one has to put the memory in normal mode first. Such a mechanism may also be driven by explicit operations in the object code of an application program, if a static analysis has identified pieces of the program during which some variables need not be accessed.

## 3 Overview of the Lustre Model

### 3.1 Principles and Main Structure

The structure of the Lustre model is inherited from the Glonemo model written in ReactiveML. Although  is a synchronous language, it is built on top of OCaml, and inherits its high-level and general-purpose features. There is no automatic translation from ReactiveML to Lustre, even for the subset of ReactiveML that matches the expressive power of Lustre (no dynamic data structures, no dynamic process creation). However, the two languages are quite close, and a systematic translation can be performed, by hand. The Lustre model is called Lussensor. Compared to Glonemo, it has been enriched with more details on the hardware of a node (DVS for the CPU, memory consumption of various kinds of memories, etc.).

The idea is to include one modeling component for each source of energy consumption we may want to model, even if we want to consider simple models where not all the sources are described in full details. Each element of the model is a data-flow box, also called *node* in Lustre. It has several input flows, several output flows, and some internal memory. Nodes are connected together as in synchronous circuits. We comment on synchrony and asynchrony in section 7. The main structure of the model is given by Figure 1.

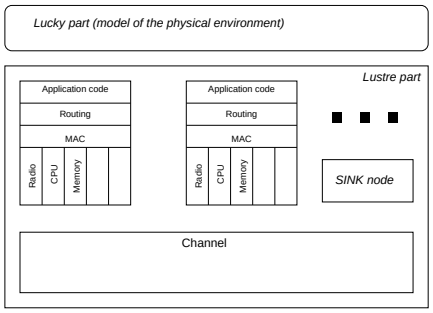


**Fig. 1.** Main Structure of the Model

The LUCKY [11, 12] part is used to model the physical environment, following the ideas described in [24]. We will concentrate on the LUSTRE part here. There are $n$ instances of the same LUSTRE node `sensor`, representing the $n$ identical sensors of a network, plus a special sink node. $n$ has to be chosen statically, but we could choose $n$ as the maximum number of sensors potentially present in the system; in this case, the model of a single sensor has an additional state "non existing", and it can be "created" or "destroyed" during the simulation.

All the components of the model are *deterministic* LUSTRE programs, although some of them need random values (e.g., the protocols). All the random values needed in the components are exposed as explicit inputs, connected to global inputs of the model, and then to an external generator. We could use a call to an external C function locally, but exposing the random value as an input is better for analysis purposes, because explicit abstractions can be made on its value.

Inside the Lustre node that represents one `sensor`, everything is *synchronous*. It is the right modeling since the physical node itself is a synchronous circuit. Between the sensors however, it is not the case. Although the physical nodes of a sensor network do have a physical clock, these clocks cannot be assumed to be synchronized during the whole lifetime of the network. Modeling the whole network is therefore one particular instance of the famous problem: *how to model asynchrony in a synchronous language?*. The general framework has been studied a lot (see [17, 5, 8]) and consists in equipping each asynchronous process with an additional input that plays the role of an activation condition for it. A specific global constraint on these activations conditions represents one special form of asynchrony. No constraint at all means pure asynchrony. A similar desynchronization mechanism is implemented in LUSSENSOR but we do not detail it in the sequel.

The model should describe what happens precisely in the communication medium, i.e., the air in which the radio transmission occurs. We could even include electromagnetic perturbations, or other similar phenomena. All these modeling aspects are grouped in a LUSTRE node called `channel` that knows about the topology of the network. When a sensor emits something with its radio, this is modeled by the corresponding LUSTRE node sending a signal to the `channel` node, which may compute which of the other nodes will hear something, depending on their relative positions, and possibly integrating perturbations of the channel.

Each *sensor* instance is structured into several components: the application software; the routing protocol (usually software); the MAC protocol (could be software or hardware, at least partially); energy models for all the significant pieces of hardware (radio, CPU, memories, sensor, etc.).

The same principle is applied for all the *energy models*: we identify a (small) set of discrete significant values for the energy consumption of the device, corresponding to its well-identified *running modes*. Then we list all the possible mode changes. Physically, these transitions between modes may take some time and energy too. For instance, switching the radio from sleeping to emitting mode has a cost, in both time and energy. We decide to encode all this phenomena into usual automata: spending time and energy is associated with states, the transitions are instantaneous and consume nothing. This means that we add some fictitious "states" to model the time and consumption of physical mode changes. Once these automata have been designed, the encoding into Lustre is very systematic.

Exploiting the various energy modes of hardware devices may be done in several ways. Our global model should provide a way to model any solution. Consequently, we provide a coordination between the model component that represents the application code, and the energy models of the CPU and the memory (see also section 4.3).

## 4 The Model Details

We then describe the components of the model in more details. The hierarchic structure of the Lustre part is described below. For each element we list the inputs and then the outputs, between parentheses. All these communications between the elements are signals, or flows, in the sense of Lustre, i.e., sequences of values over time. Technically, the node `channel` is the main node of the Lustre model, and its code contains the many instantiations of the sensor node.

The channel (sensor values, random data) (array of energies spent) =
- The sensor nodes 1, 2, ... $N$.
  Each node (sensor value, random data, radio inputs) (energy spent) =
  - Application (sensor value)
    (commands for Sensor, CPU, RAM, Flash energy models)
  - Routing
    (requests from appli, info from MAC) (requests to MAC, info to appli)
  - MAC (random data, radio input, requests from routing)
    (radio output, info to routing)
  - Sensor, CPU, RAM, Flash energy models
    (commands from appli) (energy spent)
  - Summation of the energies spent in this node
- The sink node (radio input)
- Summation of the energies spent since the beginning, for each node
- The data structures for the topology and state of the channel

### 4.1 Hardware Components

The energy models of the hardware parts are small automata, that can be encoded systematically into Lustre. The general form of the Lustre encoding can be observed on the partial model of the RAM given in Figure 2. Such a component outputs

the current energy, i.e., the energy spent during one instant of the basic clock. Some other components will gather all these values and sum them to compute the global consumption of the network. The input is a mode change request, given as the identity of the mode to reach. These components will be connected to other parts of the global model, in which the decisions for changing modes can be taken (for instance in the application software, see section 4.3).

The state is encoded by an integer or by a vector of Boolean values, depending on what we want to do with the model. For simulation purposes, it is better to use an `int`, but for validation purposes it is usually better to exhibit Boolean encodings wherever it is possible (because the exploration of the model becomes decidable). The transformation between the two forms can be done automatically in Lustre.

```
node RAM (mode_change: int) returns (energy: real);
var current_mode: int;
let
  -- Encoding of the transitions
  current_mode = RAM_MODE_OPERATE ->
          if mode_change = MODE_DONTCHANGE
          then pre(current_mode)
          else if mode_change = RAM_MODE_OPERATE
               then RAM_MODE_OPERATE
          else ...
  -- Computation of the energy spent
  energy = if  current_mode = RAM_MODE_OPERATE
           then RAM_POWER_OPERATE
           else if current_mode = RAM_MODE_STANDBY
           then RAM_POWER_STANDBY
           else ...
tel
-- somewhere else in the global model, summation of
-- the ``instantaneous'' energy values computed by RAM:
sum = 0.0 -> RAM (...) + pre (sum) ;
```

**Fig. 2.** Example Lustre encoding for an automaton modeling energy consumption (all the capitalized words are constants).

**RAM and Flash Memory** The RAM memory usually has 4 modes, and not all mode-changes are possible. The modes are: `Off`, `Idle` (the memory can be read and written to normally), `Standby` (the memory cannot be read nor written to; its consumption is low; it takes some time to put the memory in `idle` mode) and `Deep-standby` (same behavior as the previous one,it consumes even less, and it takes more time to put the memory in `idle` mode).

In the model of Figure 3, for sake of simplicity, we did not model the time needed to switch between modes, because of its very small order of magnitude, compared to other times in the global model. But it could be done easily (see the principle on the radio model). The consumptions to be attached to the states are taken from the documentation of the STMicroelectronics SRAM DS2016.

For the Flash memory (Fig. 4), it is even simpler, because it will be used to store the program to be loaded. It does not need to be written to. The modes are: `Standby` and `read`. The consumptions are taken from the documentation of the SGS-THOMSON M28F256.

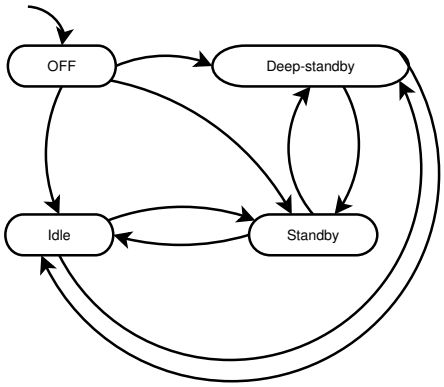


**Fig. 3.** Model of the RAM

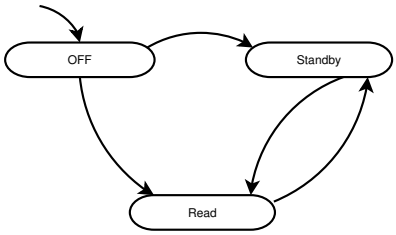


**Fig. 4.** Model of the Flash

**The CPU and the Sensor** The model of the CPU (Fig. 5) is a simple DVS model, corresponding to most existing DVS mechanisms. The model of the sensor (Fig. 6) is included in our model because we suspect that intelligent sensors have several energy modes, but we did not find appropriate documentation yet. Anyway, a sensor could have at least two modes, depending on the fact that it is activated or not. This is reflected in the simple model given here, and could be enriched to take into account a more accurate sensor documentation.

**The Radio** The radio (Fig. 7) is the most interesting energy model. The modes are: `Off`, `Idle`, `Hibernate`, `Transmit`, `Doze` and `Receive`, depending on the activity of the radio. The states denoted with dashed lines do not correspond to these modes, but they are added in order to be able to attach all timing and energy consumption information to *states*, whereas all the transitions are instantaneous and consume nothing. In the Lustre encoding, all the 10 states are represented. The information to be attached to the states is taken from the documentation of the Freescale MC13192, which implements the 802.15.4 norm.

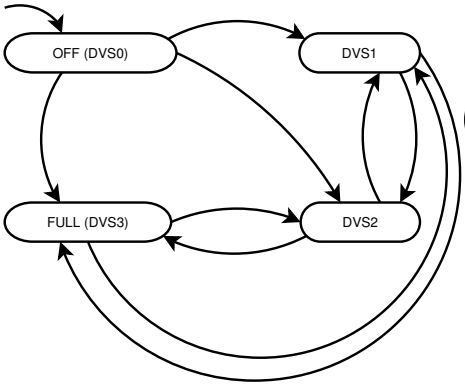


**Fig. 5.** The Model of the CPU

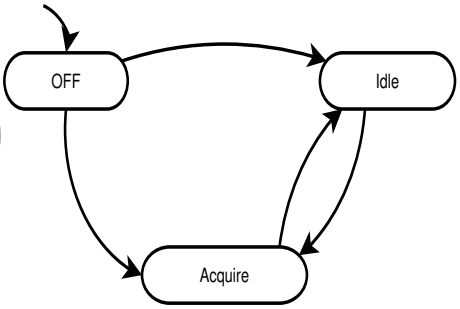


**Fig. 6.** Model of the Sensor

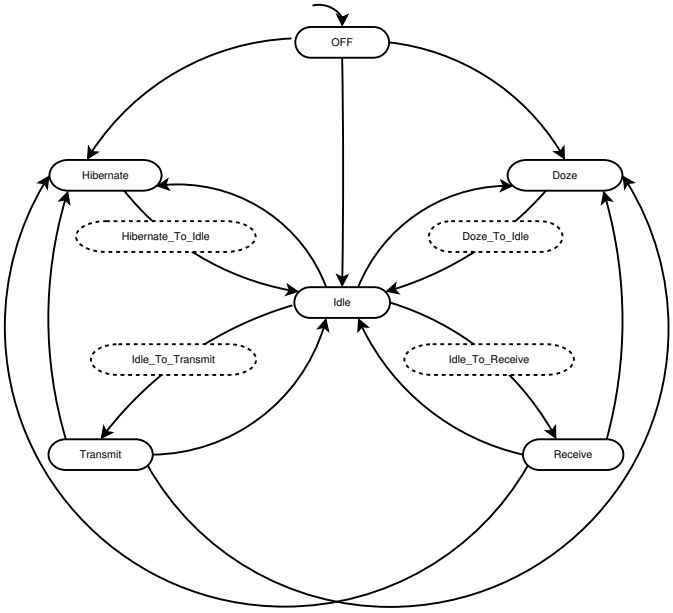


**Fig. 7.** Model of the Radio

### 4.2 Protocol Layers

In this paper we consider that the protocol layers are implemented in software. In order to include them in our global model, with the appropriate level of detail, we need to consider their object code. Indeed, when some software element drives the energy-saving mechanisms of the hardware, it is visible at the granularity level of the machine instructions. An assembly-line code can be easily described by an automaton (the control graph of the program), and that is what we do here. The automaton is then encoded into LUSTRE.

In order to give an idea of the levels of details that need to be modeled, we give a brief description of the MAC and routing algorithms included in LUSSENSOR. Annex A, page 17, is the complete LUSTRE code of the MAC component in LUSSENSOR. It needs not be read in details, but it gives an idea of the complexity of the code.

**Medium Access Control** The MAC protocol implemented in LUSSENSOR is a preamble MAC protocol (see, for instance, WiseMAC [7]). Each node periodically checks whether the channel is free. If the channel is busy, the node will let its radio on to get the packet that follows the preamble. Otherwise, it goes back to sleep mode.To avoid collisions we implement a *back-off*: the sender has to wait for a random time before emitting anything, then it scans the channel and if the channel is clear (Clear Channel Assessment, CCA), it sends the preamble and then the message. Otherwise, it delays the emission by setting a timer at random between 0 and $\mathrm{cw}_{\max}$. A preamble precedes each data packet for alerting the receiving node. All nodes in the network sample the medium with a common period.

The control automaton corresponding to this algorithm has 10 states. The LUSTRE encoding of this component will be connected to the component representing the application code, and to the component representing the channel. It also receives a random `int` value from the outside (see comment in section 3.1).

For the connection to the channel, we model the radio phenomena by a pair `(signal, packet_data)`, where `packet_data` encodes the data transmitted (a LUSTRE array of `int`s) and `signal` is a value (`int` or `Bool` encoding the elements of the set { `RF_SIGNAL_NONE`, `RF_SIGNAL_PREAMBLE`, `RF_SIGNAL_PACKET`, `RF_SIGNAL_COLLISION` }. This represents the fact that, from the point of view of the MAC, the radio is able to give the following information: either there is no signal, or there is a signal corresponding to a preamble, or there is a signal corresponding to a packet, or there is something that cannot be interpreted, meaning there is a collision. As an example, the interface of the LUSTRE node for the MAC is given by Figure 8.

**Routing** The routing protocol included in LUSSENSOR is the two-phase directed diffusion described in [10]. The sink first broadcasts an "interest" message to the whole network. The request can be *"send the temperature once an hour"*, or *"if the temperature increases sharply, send a message"* . This interest message is sent using a flooding routing mechanism: each node retransmits all the packets it receives except the ones it has already forwarded. When a node receives an interest, it checks whether it is concerned by the request and then forwards the packet. It will always send the

```
node MAC (
  -- to be left as a global input of the model:
  random_mac: int;
  -- From application code
  start_mac: bool;
  want_to_transmit: bool; -- the appl.  wants to transmit a packet
  packet_to_transmit: useful_packet_data; -- data to be trans.
  -- From channel
  rfin_signal: int; -- type of the signal received on the radio
  rfin_packet_data: packet_data -- data received
) returns (
 energy: real;
 -- To application code
 busy: bool; -- the MAC is busy, cannot transmit now
 packet_received: bool; -- MAC has received a packet
 packet_transmitted: bool;  --  MAC has transmitted a packet
 received_data: useful_packet_data; -- the packet received
 -- To Channel
 rfout_signal: int; -- type of the signal emitted on the radio
 rfout_packet_data: packet_data; -- data emitted );
```

**Fig. 8.** Interface of the MAC component

values through the route that was used to reach it from the sink. The algorithm is encoded into an automaton, and then in Lustre.

### 4.3 The Application code and the Model of the Channel

The application code is a simple algorithm that emits the value sensed on a regular basis. It has 8 control states, and computes the commands `mode_sensor`, `mode_cpu`, `mode_flash` and `mode_ram` to be connected to the corresponding inputs of the hardware device models. For the moment, the values of these commands are entirely defined by the control state. The effect of any static analysis that would insert such commands in the object code of the application can be easily included in our model.

In Lussensor, the channel is the part of the global model that takes care of the air where communications take place, and knows about the topology of the network. The corresponding Lustre node `channel` computes which nodes receive a correct signal, which nodes are jammed, etc. It is quite complex because the main algorithms involved are iterative algorithms on matrices, which have to be encoded into the Lustre-V4 array operators (originally defined for circuit design, and based on static recursion, see [23]). But there is no intrinsic difficulty here.

## 5 Related Work

The first category of "virtual prototyping" approaches corresponds to the definition of formal models for performance analysis. These models are usually quite simple,

and this is the reason why they are used mainly to compare protocols on one link. Since Kleinrock and Tobagi [13], they have been used extensively for the evaluation of MAC protocols. However, they cannot be used to compare complete protocol stacks.

All other virtual prototyping approaches are developed in order to include some description of the network behavior in the model. This gives more complex models, of course, for which there is no simple set of equations that could be solved. These models may be used for simulation, but we can also hope to use then for formal validation, if they are described in well-defined languages or formalisms.

Because network simulators are extensively used in the network community research, many relevant simulators have been developed. NS-2 [2] is a packet-level simulator that was first designed for wired networks. NS-2 is a discrete event simulator. The interest of having one single simulator is to enable comparisons between different protocols without the need to implement the protocol we want to compare with. Indeed, NS offers a large protocol library. However, NS is not really scalable: it is convenient for simulating a few hundred nodes only. Because one of the key issues in sensor networks is power consumption, people began to develop simulators that take the energy consumption into account.

Avrora [27] is written in Java and is cycle-accurate. It is able to execute the binary code of an application. The efficiency of the simulation relies on a quite complex synchronization pattern which in fact constitutes the model of the radio. For the environment, models are still needed, and the interaction between a model of some component and the exact description of another component is not formalized. It would be hard to use this framework to play with various abstractions.

Atemu [22] executes binary code and synchronizes the nodes on the clock cycle of the processor. Fine grain properties can be obtained up to 120 nodes. To our opinion, simulating the hardware at this level of detail is probably hopeless.

TOSSIM [18] is the simulator dedicated to TinyOS [26] applications. TOSSIM does not provide a model of the consumption. To overcome this limitation, it has been extended with PowerTOSSIM [25]. In PowerTOSSIM, each state of the CPU, Radio and EEPROM is associated with a cost. Running the simulator computes the energy consumption of each node.

AEON [14] proposes to build an energy model by running a real network, and then to include this model in a simulator like AVRORA, to do some profiling. AEON allows to observe the impact of the energy management primitives of TinyOS.

None of these simulators uses a formal model that could be used for validation.

On the other hand, the formal validation community does not seem to have started working specifically on sensor networks. To our knowledge, there is no other approach for the formal and global modeling of sensor networks, for which we can hope to use validation tools. Some experiments in modeling and analyzing sensor networks have been made with tools like HyTech [9] or Uppaal [15], but the models are still very abstract.

## 6 Current Uses of the Model

### 6.1 Validation by Simulations

The model has been developed progressively, and each component tested before integration. The complete model has been simulated with the LUSTRE interpreter, but it is quite slow (even for a small number of sensors, typically 10), and the graphical interface is poor, compared with what we can do in REACTIVEML. The intended use is as follows: for any energy-related property one would want to observe on the model, design a LUSTRE *observer* (a special node that may read all the values of the input, output and local variables, but has no effect on the behavior) that outputs numbers; compile the LUSTRE model together with the observer; connect the code to the environment model in LUCKY (this part generates values for the sensors, and also the random values needed by the protocol parts); run this a large number of times, storing the outputs; draw curves from the output sequences.

This method makes the LUSTRE model comparable to tools like `ns2`. We are currently investigating the "observer" version of the main quantitative evaluations usually found in the papers of the network community.

### 6.2 Uses of Lucky models

LUCKY [11, 12] belongs to the LUSTRE toolbox. It allows to describe non-deterministic reactive behaviors as sets of parallel communicating automata with weights representing probabilities. A LUCKY component may be used in our global model to replace any of the LUSTRE components, provided it has the same input/output interface.

The first use of LUCKY is to model the physical environment, i.e., the non-deterministic process that generates spatially and temporally correlated stimuli for the sensors. This is the same approach as in [24].

The next thing we will do is to use LUCKY to model perturbations in the channel (shadowing, fading, path-loss). We'll have to encode in LUCKY the accurate probabilistic modelings that have been proposed for these phenomena in the network community. The LUSTRE model is an appropriate platform for these experiments.

A similar use of LUCKY would be to replace a part of the network (a subset of the nodes) by a *traffic generator*, i.e., a non-deterministic process that generates the states of the channel for the remaining nodes. This is related to the next point, since it is a way of abstracting the global model.

### 6.3 Modular Abstractions and Formal Analyzes

As mentioned in the introduction, analyzing formal models of sensor networks means we are able to perform quite drastic abstractions, but these abstractions may depend on the kind of property to be analyzed. We are interested in properties that talk about the energy consumption, for instance: "*is it possible to spend more than energy E in less that time T?*". This is a safety property. The LUSTRE model in which both time and energy are encoded into some numbers that behave as counters may

theoretically be fed into a verification tool that deals with numbers symbolically (abstract interpretation for instance). But the model for thousands of nodes is huge.

We propose to use the LUSTRE model as a modular-abstraction framework. The idea is the following: replacing a component $C$ in a global model $M$ by a more abstract version $C'$ should yield a new global model $M'$ which is indeed more abstract than $M$. This abstraction preservation property is essential when playing with various abstractions of the individual components. We should also be able to prove the property: $C'$ is more abstract than $C$.

For the components modeling the energy consumption, the notion of abstraction has to be defined precisely. In such a model $C$, the energy attached to a "state" is in fact a worst-case estimation of the energy spent by unit of time while the system is in this state. Such a model, reacting to an input sequence $I$, produces a sequence of these "instantaneous" energies, than can be summed up. Let us note $\Sigma(C, I)$ the sum of the energy outputs produced when $C$ reacts to $I$. A model $C'$ is more abstract than $C$ iff: for all $I$, $\Sigma(C, I) \geq \Sigma(C', I)$. "More abstract" means that the worst-case estimation is less precise, hence greater.

We are currently experimenting various abstract-interpretation tools to help verify automatically that an energy model $C'$ is more abstract than a model $C$. The case study is the model of the radio (see Figure 7), for which various approximations can be derived.

## 7 Conclusions and Perspectives

We have designed the architecture of a global and accurate model of sensor networks, in LUSTRE. All the elements are taken into account, except the operating system. We could have included one more component for the OS without difficulty, but a lot of WSN solutions are considering static scheduling instead of using an OS, which is probably a good choice for energy consumption. Hence the "application" component of our model is sufficient. The software parts may be included at the level of detail of machine instructions, which gives a fine-grain modeling of energy consumption. Any hardware device can be modeled by a dedicated energy-model, as we did for the radio, the CPU, the memories, and the sensor. The values taken form the data-sheets of current technology devices can be directly included in our models.

We think that our model is as precise as the cycle-accurate models obtained with tools like Avrora or Atemu (see related work). LUSSENSOR is not intended for debug simulations (it does not provide graphical outputs), but the compiled code may be used for batch simulations. Moreover, we think that LUSSENSOR has several other important qualities:

- LUSSENSOR is a modular model that may serve as a common platform for several abstractions. Moreover, the various abstractions can be compared, thanks to the abstraction partial order on the energy components, as described above.
- Adding observers to compute quantitative measures of the network behavior is very easy

– LUSTRE being a declarative language, the global model is essentially a set of Boolean and numerical equations, for which we can hope to use a large set of symbolic verification tools.
– The LUSSENSOR platform may be used to include existing probabilistic models as components, if we are able to describe them in LUCKY.

LUSSENSOR is a first step, for which the main perspectives are the following. First, we will use LUSSENSOR as a case-study for our "modular worst-case energy models" approach; second, we will investigate the combined use of LUCKY and LUSSENSOR to design performance models of sensor networks that contain some details on the behavior of the computing parts. Indeed, as mentioned in section 5, the mathematical models used for the performance evaluation of protocols are too simple when it comes to representing complete protocol stacks or complex radio channel behaviors (i.e., collisions).

## References


1. Coronis. Internet address: http://www.coronis-systems.com/.
2. The Network Simulator - ns2. http://www.isi.edu/nsnam/ns/.
3. Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325–349, May 2005.
4. Ian F. Akyildiz, W. Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.
5. Paul Caspi, Christine Mazuet, and Natacha Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. In Udo Voges, editor, *Computer Safety, Reliability and Security, 20th International Conference, SAFECOMP 2001, Budapest, Hungary, September 26-28, 2001, Proceedings*, volume 2187 of *Lecture Notes in Computer Science*, pages 215–226. Springer, 2001.
6. Victor Delaluz, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Anand Sivasubramaniam, and Mary Jane Irwin. Hardware and software techniques for controlling DRAM power modes. *IEEE Trans. Computers*, 50(11):1154–1173, 2001.
7. Christian C. Enz, Amre El-Hoiydi, Jean-Dominique Decotignie, and Vincent Peiris. Wisenet: An ultralow-power wireless sensor network solution. *IEEE Computer*, 37(8):62–70, 2004.
8. N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, Grenoble, October 2002. LNCS 2491, Springer Verlag.
9. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to hytech. Technical Report TR95-1532, Cornell University, Computer Science Department, August 29, 1995.
10. Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MOBICOM*, pages 56–67, 2000.
11. E. Jahier, P. Raymond, and P. Baufreton. Case studies with lurette v2. *International Journal on Software Tools for Technology Transfer (STTT)*, 2006.
12. E. Jahier, P. Raymond, and Y. Roux. Describing and executing random reactive systems. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, Pune, India, 2006.
13. Leonard Kleinrock and Fouad A. Tobagi. Packet switching in radio channels: Part i–carrier sense multiple-access modes and their throughput-delay characteristics. In *IEEE Transactions on Communications*, volume 23, pages 1400–1416, december 1975.

14. O. Landsiedel, K. Wehrle, and S. Gtz. Accurate prediction of power consumption in sensor networks. In *Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, 2005.
15. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
16. Hyung Gyu Lee and Naehyuck Chang. Energy-aware memory allocation in heterogeneous non-volatile memory systems. In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 420–423, New York, NY, USA, 2003. ACM Press.
17. P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.
18. Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA. ACM Press.
19. Louis Mandel and Marc Pouzet. Reactive ML.
20. Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, April 2005. ENTCS.
21. Sylvain Plancoulaine. Architecture logicielle-matrielle pour protocole mac dans un rseau de capteurs. Rapport de stage de master de recherche csina, 5 mois, encadrant a. bachir, Université Joseph Fourier, Marseille, 2006.
22. Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, and John S. Baras. ATEMU: A Fine-grained Sensor Network Simulator. *Secon*, 2004.
23. F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits, a hardware implementation of LUSTRE. In REX *Workshop on Real-Time: Theory in Practice, DePlasmolen (Netherlands)*, pages 195–208. LNCS 600, Springer Verlag, June 1991.
24. Ludovic Samper, Florence Maraninchi, Laurent Mounier, Erwan Jahier, and Pascal Raymond. On the importance of modeling the environment when analyzing sensor networks. In *Proceedings of International Workshop on Wireless Ad-Hoc Networks 2006 (IWWAN 2006)*, page 7, New York, United States, June 2006.
25. Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *SynSys*, pages 188–200, 2004.
26. TinyOS Team. Tinyos.
27. Ben L Titzer, Daniel K Lee, and Jens Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. *Proceedings of IPSN*, 2005.
28. Lin Yuan and Gang Qu. *Energy-Efficient Design of Distributed Sensor Networks*, chapter 38. CRC press, October 2004.

# A The MAC protocol

```
-- Model for the mac protocol of a sensor
-- Based on Preambule Sensing
-- (see MAC protocols for Wireless sensor Network: a survey)

include "definitions.lus"
include "radio.lus"

-- Duration of the sleeping time, here 10ms
const MAC_SLEEPING_TIME = 0.010;
-- Duration of the carrier sensing time (200s)
const MAC_CARRIER_SENSING_TIME = 0.000200;
-- Duration of the preamble time (sleep + carrier sense + radio awakening time):
const MAC_PREAMBLE_TIME = MAC_SLEEPING_TIME + MAC_CARRIER_SENSING_TIME +
      RADIO_TRANS_DOZE_TO_IDLE + RADIO_TRANS_IDLE_TO_RECV;
-- Backoff slot time (100s):
const MAC_BACKOFF_SLOT_TIME = 0.000100;


-- Number of sleeping cycles:
const MAC_SLEEPING_CYCLES        = int(ceil(MAC_SLEEPING_TIME / TIME_SCALE));
-- Number of carrier sensig cycles
const MAC_CARRIER_SENSING_CYCLES = int(ceil(MAC_CARRIER_SENSING_TIME / TIME_SCALE));
-- Number of preamble sending cycles
const MAC_PREAMBLE_CYCLES        = int(ceil(MAC_PREAMBLE_TIME / TIME_SCALE));


-- State of the constrol automaton of the MAC Protocol
const MODE_MAC_INIT              = 1;
const MODE_MAC_SLEEP             = 2;
const MODE_MAC_CARRIER_SENSE     = 3;
const MODE_MAC_RECEIVE_PREAMBLE  = 4;
const MODE_MAC_RECEIVE_PACKET    = 5;
const MODE_MAC_BACKOFF           = 6;
-- CCA: Clear Channel Assessment
const MODE_MAC_CCA               = 7;
const MODE_MAC_PREAMBLE_EMISSION = 8;
const MODE_MAC_PACKET_EMISSION   = 9;

node MAC (-- From environment
          random_mac: int;
          -- From appli
          start_mac: bool;
          want_to_transmit: bool;
          packet_to_transmit: useful_packet_data;
          -- From channel
          rfin_signal: int;
          rfin_packet_data: packet_data
          )
```

```
returns (energy: real;
         -- To appli
         busy: bool;
         packet_received: bool;
         packet_transmitted: bool;
         received_data: useful_packet_data;
         -- To Channel
         rfout_signal: int;
         rfout_packet_data: packet_data;
);

var
  -- To radio
  mode_change: int;
  out_signal: int;
  out_packet_data: packet_data;
  -- From radio
  in_signal: int;
  in_packet_data: packet_data;
  rf_active: bool;
  radio_busy: bool;

  -- Locals
  s_random: int;
  state: int;
  -- Counters
  sleeping_counter: int;
  sensing_counter: int;
  backoff_counter: int;
  emission_counter: int;
  -- Packets
  tx_cycles: int;
  tx_packet_buffer: packet_data;
  tx_packet_size: int;
let
  -- Sample random to avoid it changing
  s_random = random_mac -> if want_to_transmit then random_mac
                           else pre(s_random);

  -- State management
  state = MODE_MAC_INIT ->   -- Init mode; do nothing
          if pre(state) = MODE_MAC_INIT then
            -- Start MAC when software is ready
            if pre(start_mac) then MODE_MAC_SLEEP
            else MODE_MAC_INIT


          -- Sleep Mode: wait for beeing awakened or to
          -- have a packet to transmit
```

```
else if pre(state) = MODE_MAC_SLEEP then
       -- Carrier Sense has priority over transmission
       if pre(sleeping_counter) = MAC_SLEEPING_CYCLES
       then MODE_MAC_CARRIER_SENSE
       else if want_to_transmit
            then MODE_MAC_BACKOFF
            else MODE_MAC_SLEEP

-- Carrier sensing: sense if a preamble is on the channel
else if pre(state) = MODE_MAC_CARRIER_SENSE then
       -- Back to sleep state if there is a collision
       -- or if the carrier sensing time is ellapsed
       if pre(in_signal) = RF_SIGNAL_COLLISION or
          pre(sensing_counter) = MAC_CARRIER_SENSING_CYCLES
       then MODE_MAC_SLEEP
       -- If there's a preamble detected on the channel,
       -- go in the receive mode
       else if pre(in_signal) = RF_SIGNAL_PREAMBLE
       then MODE_MAC_RECEIVE_PREAMBLE
       -- Don't accept direct packet data
       else if pre(in_signal) = RF_SIGNAL_PACKET
       then MODE_MAC_SLEEP
       else MODE_MAC_CARRIER_SENSE

-- Receive Preamble mode: preamble detected, receive end of preamble
else if pre(state) = MODE_MAC_RECEIVE_PREAMBLE then
       -- Transition to receive packet mode if packet data coming
       if      pre(in_signal) = RF_SIGNAL_PREAMBLE
       then MODE_MAC_RECEIVE_PREAMBLE
       else if pre(in_signal) = RF_SIGNAL_PACKET
       then MODE_MAC_RECEIVE_PACKET
       -- Error, return to sleep mode
       else MODE_MAC_SLEEP

-- Receive Packet Mode: preamble detected, receive
-- end of preamble & following packet
else if pre(state) = MODE_MAC_RECEIVE_PACKET then
       -- Receive packet data as long as possible (signal is packet)
       if pre(in_signal) = RF_SIGNAL_PACKET
       then MODE_MAC_RECEIVE_PACKET
       else MODE_MAC_SLEEP

-- Backoff: wait for max(time for radio being ready,
-- random times MAC_BACKOFF_SLOT_TIME)
else if pre(state) = MODE_MAC_BACKOFF then
       -- Go to CCA only when backoff is done, and radio is ON
       if pre(backoff_counter) >=
          int(ceil((MAC_BACKOFF_SLOT_TIME * real(s_random)) / TIME_SCALE))
          and pre(rf_active)
       then MODE_MAC_CCA
```

```
               else MODE_MAC_BACKOFF

        -- CCA: check if the medium is busy before setting sending mode and packet
        else if pre(state) = MODE_MAC_CCA then
               -- If channel is busy, return to sleep mode
               if pre(in_signal) <> RF_SIGNAL_NONE then MODE_MAC_SLEEP
               -- Else, go to preamble emission state, when radio
               -- has finished returning to idle mode
               else if pre(pre(state)) = MODE_MAC_CCA
                       and pre(radio_busy) = false
                    then MODE_MAC_PREAMBLE_EMISSION
                    else MODE_MAC_CCA

        -- Preamble emission: send a preamble during MAC_PREAMBLE_EMISSION_TIME
        else if pre(state) = MODE_MAC_PREAMBLE_EMISSION then
               -- Emit packet when preamble done
               if pre(emission_counter) >= MAC_PREAMBLE_CYCLES
               then MODE_MAC_PACKET_EMISSION
               else MODE_MAC_PREAMBLE_EMISSION

        -- Packet emission: send the packet
        else if pre(state) = MODE_MAC_PACKET_EMISSION then
               -- Return to sleep mode when packet transmitted
               if pre(emission_counter) > pre(tx_cycles) then MODE_MAC_SLEEP
               else MODE_MAC_PACKET_EMISSION

             else MODE_ERROR;


-- Duration of sleep period during carrier senses
sleeping_counter = 0 ->
      if state = MODE_MAC_SLEEP
      then pre(sleeping_counter) + 1
      else 0;

-- Duration of carrier sense; start to count
--  only when RF is really active
sensing_counter = 0 ->
     if state = MODE_MAC_CARRIER_SENSE and rf_active
     then pre(sensing_counter) + 1
     lse 0;

-- Duration of transmission; start to count only
-- when RF is really active
emission_counter = 0 ->
     if (state = MODE_MAC_PREAMBLE_EMISSION or state = MODE_MAC_PACKET_EMISSION)
        and rf_active
     then pre(emission_counter) + 1
     else 0;
```

```
-- Duration of backoff period
backoff_counter = 0 ->
     if state = MODE_MAC_BACKOFF
     then pre(backoff_counter) + 1
     else 0;

-- We use the doze mode for sleeping time
-- (hibernate mode has a too long awakening time)
mode_change = RADIO_MODE_DOZE ->
     -- Don't change radio mode as Software is uninitialized
           if state = MODE_MAC_INIT then MODE_DONTCHANGE

           else if state = MODE_MAC_SLEEP then
             -- Change radio mode only once (to avoid spurious mode changes)
             if pre(state) = MODE_MAC_SLEEP then MODE_DONTCHANGE
             else RADIO_MODE_DOZE

           -- Awakening of the radio, transition to idle, and then receive mode
           else if state = MODE_MAC_CARRIER_SENSE or state = MODE_MAC_BACKOFF then
                  -- First put radio in Idle mode
                  if      pre(state) = MODE_MAC_SLEEP then RADIO_MODE_IDLE
                  -- When radio is in idle mode; put it into receive mode
                  else if pre(radio_busy) = false then RADIO_MODE_RECEIVE
                  -- Else don't change radio mode (to avoid spurious mode changes)
                  else MODE_DONTCHANGE

           -- Receive mode:  don't change anything since radio is already receiving
           else if state = MODE_MAC_RECEIVE_PREAMBLE or
                   state = MODE_MAC_RECEIVE_PACKET
                then MODE_DONTCHANGE

           -- CCA: Put radio in idle mode during second CCA cycle
           -- (first cycle is for sensing)
           else if state = MODE_MAC_CCA then
                  -- Second CCA cycle -> go to idle mode
                  if pre(state) = MODE_MAC_CCA then RADIO_MODE_IDLE
                  -- First CCA cycle -> sensing -> stay in receive mode
                  else MODE_DONTCHANGE

           -- Emission d'un paquet ou preambule
           else if state = MODE_MAC_PREAMBLE_EMISSION or
                   state = MODE_MAC_PACKET_EMISSION then
                  -- Put radio once for all (to let radio automatically
                  -- return to idle mode when transmission is done)
                  -- in good mode if ready (it's in idle state for now)
                  if pre(radio_busy) = false
                     and pre(state) <> MODE_MAC_PREAMBLE_EMISSION and
                     pre(state) <> MODE_MAC_PACKET_EMISSION
                  then RADIO_MODE_TRANSMIT
                  else MODE_DONTCHANGE
```

```
               else MODE_ERROR;

-- Send to the radio the type of RF signal to send
out_signal = RF_SIGNAL_NONE ->
    if state = MODE_MAC_PREAMBLE_EMISSION then RF_SIGNAL_PREAMBLE
    else if state = MODE_MAC_PACKET_EMISSION then RF_SIGNAL_PACKET
    else RF_SIGNAL_NONE;

-- Packet to send to the radio
-- 8 bytes of header * (32s each) + payload * (32s/byte)
tx_cycles = 0 ->
   if want_to_transmit
   then MAC_PREAMBLE_CYCLES +
        int(ceil(real(256 + (tx_packet_size * 32)) / TIME_SCALE * 0.000001))
   else pre(tx_cycles);
tx_packet_size = 0 ->
       if mode_change = RADIO_MODE_TRANSMIT
       then PACKET_MAX_PAYLOAD      -- Must be modified to
                                    -- support varying packet lengths
       else pre(tx_packet_size);
tx_packet_buffer = PACKET_DEFAULT ->
       if want_to_transmit then MAKE_PACKET(packet_to_transmit)
       else pre(tx_packet_buffer);

-- RF output management
-- Emit packet data only during last cycle of emission
out_packet_data = PACKET_DEFAULT ->
       if state = MODE_MAC_PACKET_EMISSION and
          pre(emission_counter) = pre(tx_cycles)
       then tx_packet_buffer
       else PACKET_DEFAULT;
packet_transmitted = false ->
       (pre(state) = MODE_MAC_PACKET_EMISSION
       and state = MODE_MAC_SLEEP);

-- Packet received from radio
packet_received = false ->
       (pre(state) = MODE_MAC_RECEIVE_PACKET
        and pre(in_signal) = RF_SIGNAL_PACKET
        and pre(PACKET_VALID(in_packet_data)));


-- A packet is present
received_data = DATA_DEFAULT ->
       if packet_received
       then EXTRACT_PACKET_DATA(pre(in_packet_data))
       else DATA_DEFAULT;

-- MAC state
```

```
  busy = ((state <> MODE_MAC_INIT and state <> MODE_MAC_SLEEP)
         or mode_change <> MODE_DONTCHANGE);

  -- Radio node
  (energy, in_signal, in_packet_data, rf_active,
   radio_busy, rfout_signal, rfout_packet_data) =
   RADIO (rfin_signal, rfin_packet_data,
          mode_change, out_signal, out_packet_data);
tel
```