A Generic Structure for Modeling Time and Energy Consumption in Abstract Virtual Prototypes of Embedded Systems

Florence Maraninch and Catherine Parent-Vigouroux and Karel Heurtefeux and Pascal Raymond

Verimag Research Report n^o TR-2011-17

November 2011

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

A Generic Structure for Modeling Time and Energy Consumption in Abstract Virtual Prototypes of Embedded Systems

Florence Maraninch and Catherine Parent-Vigouroux and Karel Heurtefeux and Pascal Raymond

November 2011

Abstract

A *virtual prototype* is an *executable* model of the hardware platform on which the software can be developed. This allows an early evaluation of the functional correctness and performances of the whole system. Among virtual prototypes, *emulators*, or cycle-accurate hardware models, are close to the real hardware; their design requires a significant amount of work. There is a need for higher-level models. We propose a generic structure for the modeling of *time and energy consumption* at abstract levels, for sensor networks. We use an existing MAC protocol to show that our model is far simpler than cycle-accurate emulators, yet retaining the essential aspects of time and energy consumption.

Keywords: virtual prototyping, simulation, formal models, embedded software, energy consumption, sensor networks

Reviewers: Laurent Mounier

Notes: This work has been partially supported by the French ANR Project ARESA2 (ANR-09-VERS-017).

How to cite this report:

```
@techreport {TR-2011-17,
    title = {A Generic Structure for Modeling Time and Energy Consumption in Abstract Virtual Prototypes of Embedded Systems},
   author = {Florence Maraninch and Catherine Parent-Vigouroux and Karel Heurtefeux and Pascal Raymond },
   institution = {{Verimag} Research Report},
   number = {TR-2011-17},
   year = {}
}
```

A Generic Structure for Modeling Time and Energy Consumption in Abstract Virtual Prototypes of Embedded Systems

Florence Maraninch and Catherine Parent-Vigouroux and Karel Heurtefeux and Pascal Raymond

November 2011

Abstract: A *virtual prototype* is an *executable* model of the hardware platform on which the software can be developed. This allows an early evaluation of the functional correctness and performances of the whole system. Among virtual prototypes, *emulators*, or cycle-accurate hardware models, are close to the real hardware; their design requires a significant amount of work. There is a need for higher-level models. We propose a generic structure for the modeling of *time and energy consumption* at abstract levels, for sensor networks. We use an existing MAC protocol to show that our model is far simpler than cycle-accurate emulators, yet retaining the essential aspects of time and energy consumption.

1 Introduction

The design of modern embedded systems, and of *networks of* embedded systems, has to face the increasing complexity and unpredictability of the hardware execution platforms, the need to take *non-functional properties* (e.g., energy consumption) into account as an additional optimization criterion, the complexity of the software, from the low level operating system to the application. The design of an optimal solution is often out of reach.

Design methods therefore rely on *virtual prototypes* (VPs). A VP is an *executable model* of the hardware, which can be used in order to simulate the software and evaluate the functionality and performances of the whole system early in the design cycle (before the real hardware is available).

A first example is the notion of a *virtual platforms* for systems-on-a-chip, often written in SystemC [20], at a level of abstraction called *transaction-level-modeling (TLM)* [11].

Another example is taken from the domain of telecommunication networks, where new protocols are designed, and their properties evaluated, by executing them with network simulation tools. A network simulator (e.g., Omnet++ [1]) implements a VP. An interesting sub-domain is that of wireless sensor networks (WSNs), where energy consumption plays a crucial role; in this case one can use a network of emulators as a VP (e.g., in WSNET/WSIM [9] or COOJA [8]).

For digital systems connected to some physical environment, the VP may include a model of this physical environment. This is the case when a control algorithm is designed in Simulink. This is also the case when a protocol in a wireless sensor network is designed and simulated with a model of the radio channel. In the sequel, we will not distinguish between a hardware VP, and a hardware+physics VP. Both are meant to behave as the execution environment of the software.

1.1 Main Issues for the Development of VPs

Timing: The VP should not be more *synchronized* than what the real hardware+physics can provide. A model of a distributed system in which the hardware clocks are considered as synchronized is overly optimistic. But a model where the clocks are considered to be fully *asynchronous* is not *usable*: the VP exposes too many behaviors to the software, among which very unrealistic ones. The appropriate VP in this case is a model in which the physical drift between hardware clocks is modeled. Most network simulators allow to specify different birth dates for the nodes, but offer no specific support for modeling clock drift.

Energy Consumption: There are essentially two categories of models. *Direct* models are based on so-called *power-states* models, i.e., a representation of the various operating modes of a hardware device, associated with an instantaneous consumption. For instance, the radio device in a sensor node has at least three modes: Sleep, Transmit, Receive. In Sleep mode the consumption per unit of time is far less than

in the two other modes. This is the same for a processor with dynamic voltage and frequency scaling (DVFS). Using such direct models requires a precise modeling of the hardware/software interface: another part of the model has to "drive" the power-state automaton, depending on the functional behavior. *Indirect models* can be used in more abstract settings, where the hardware and the hardware/software interface are not detailed. For instance, a lot of network models are based on the following abstraction: "*sending one bit costs k units of energy*". Validating the constant k can be quite hard.

Abstraction Level: Cycle-accurate models mimick the behavior of the hardware, and are usually considered faithful. However, developing a cycle-accurate model of the hardware platform takes a lot time. There is a need for models that are more abstract than full emulators or cycle-accurate models, yet retaining the essential timing properties of the hardware/software interface and the associated energy consumption properties.

Assessment of Accuracy: The predictions of a virtual prototype have to be compared with measures on some real platform. In the case of sensor networks, this is not easy, because measures are intrusive. The development of the SensLab platform (see `www.senslab.info`) addresses this problem, real sensor nodes being coupled with "spy" nodes.

Simulation Speed: Abstract models are likely to simulate faster than full emulators. However, most network simulators are based on a discrete-event simulation engine, with a single notion of time. Building a realistic VP with clock drift can be done by specifying "clocks" as sequences of *events*. But then, there are many such events to be trated by the simulation engine, which degrades the performances.

1.2 Contributions and Structure of the paper

The contribution of the paper is a generic structure for virtual prototypes that: (i) are detailed on the hardware/software interface, modeling time and energy consumption directly; (ii) are far simpler than full emulators; (iii) allow simulations in reasonable time. The main application is the modeling of sensor networks.

We describe the proposed generic structure, and its implementation in a synchronous language (there exist earlier experiments with synchronous languages as modeling languages for WSNs [22, 17], but they do not detail the influence of hardware clocks. We choose the language Lustre [5] because it allows an easy componentization of the model, and there are available compilers that produce efficient code for simulation. The protocols can also be written into Lustre quite easily.

This contribution is the first compulsory step to build a framework in which the accuracy of the abstract model can be assessed by comparing its predictions to actual measures. Indeed, the same specification of the protocol in Lustre can be: (i) compiled together with the hardware and channel models, for the simulation framework; (ii) or compiled into C and linked with the drivers to be deployed on the SensLab platform. By logging all the events that are non-deterministic (e.g., which nodes received a particular message, which link failed), a real scenario will be replayed on the Lustre model, for meaningful comparisons.

Section 2 lists related work; Section 3 describes the case-study; Section 4 is a short introduction to synchronous programming in Lustre, necessary for the presentation of our generic model; Section 5 presents our Lustre model of the case-study, insisting on its generic traits. Section 6 describes the experiments done with the case-study. Section 7 concludes.

2 Related Work

2.1 Synchronous Modeling of Asynchrony

Using a synchronous formalism to model asynchronous systems dates back to Milner [19]. It has been applied a lot using synchronous languages [14, 10]. In a synchronous model, time is nothing but an additional input, and there may be several such inputs to model various notions of time (e.g., non synchronized clocks). This also has the advantage that these "time" inputs need not be completely uncorrelated. For instance, in some embedded systems that use several processors for fault-tolerance, the design and implementation of control systems heavily exploits a constraint on the clock drift between the processors' clocks

called *quasi-synchrony* [6]. To summarize, when modeling asynchrony with a synchronous formalism, it is possible to cover the full range of systems, between purely synchronous systems and purely asynchronous systems, just be expressing a constraint on several inputs.

2.2 Transaction-Level Modeling

Transaction-level modeling (TLM) [11] has emerged for the virtual prototyping of systems-on-a-chip. TL models are far simpler than cycle-accurate ones; they allow very fast simulations of the embedded software, and they can be made available very early in the design cycle. Yet, the hardware/software interface is precise in TL models. This is the same goal we are pursuing for sensor networks.

The standard of the domain is SystemC/TLM [20] which provides threads, simulated time, and a discrete-event execution engine. There is a single notion of time and an instruction `wait(t)`, meaning "*wait* `t` *units of simulated time*". Using `wait(...);` in several parallel processes has very strong synchronization effects. This has been recognized as a problem. As a solution, TL models can be *approximately* timed models [7] with the instruction `wait ([d1, d2])`, meaning that a random delay in the interval `[d1, d2]` will be chosen during simulation. This makes the model non-deterministic, and is a way of avoiding spurious synchronizations between parts of the model that represent non-synchronized physical entities.

2.3 Models for energy consumption

Power-state models (see, for instance [3]) are commonly used to model energy consumption. They indicate how many units of energy are spent per unit of time, in each functioning mode of a device (e.g., for a radio component, *emitting*, *receiving*, or *idle* modes). The providers of hardware for sensor networks, like radio components, provide such power-states models (e.g., page 42 of [24]). Formally, power-state models are automata whose states are labeled by instantaneous energy consumptions of the form: $de/dt = k$, where k is a constant. The transition labels represent inputs. For a given sequence of such inputs, the automaton goes through a sequence of states $X_0X_1...X_n$. The total energy consumed is computed as: $\sum$(time spent in $X_i \times k_{X_i}$). Power-state models are also very similar to the *linear-priced timed automata (LPTA)* used in, e.g., [2].

For sensor networks, a less precise model is often used. The idea is to evaluate the energy consumption by counting the bits transmitted, using a Joule-per-bit estimation [12]. The consumption of the devices other than the radio is not taken into account. Moreover, the radio devices usually have a *sleep* mode that consumes less than the *receive* mode. A lot of MAC protocols are designed to minimize the periods of *overhearing* (when a node is in *receive* mode, while it could be in *sleep* mode). With the Joule-per-bit models, overhearing is not modeled; these models cannot be used to evaluate MAC protocols that try and reduce overhearing.

2.4 Network simulation tools vs sensor node emulators

For a review of network simulators, and their specialization for sensor networks, see [18]. If the modeling of energy consumption uses the Joule-per-bit abstraction mentioned in section 2.3, ordinary network simulators can be used to count messages and obtain energy consumption.

Otherwise, dedicated simulation tools have to be developed. In TOSSIM [16], Avrora [25], or WSIM [9], the models of the hardware are *cycle-accurate*. They may include an instruction-set-simulator for a particular micro-controller, and a precise emulator of the rest of the hardware (bus, memory, radio chip). In WSIM/WSNET, for instance, simulating a network can be done by running several WSIM node emulators in parallel, playing the real software, together with a model of the radio channel. The WSNET simulator allows to have different birth dates for nodes, but does not model clock drift.

The choice is between: (i) high-level models, with the Joule-per-bit abstraction, and a simple model of the hardware; (ii) networks of node *emulators*, modeling all the details of a particular hardware. In both cases, the models do not provide support for modeling clock drift.

2.5 Formal models applied to WSNs

A lot of formal frameworks have been applied to the validation of protocols for sensor networks, like PVS in [4], or timed Petri nets in [15]. Only a few consider energy consumption. The work published in [23] is very close to ours. It describes a formal model for the analysis of clock synchronization in a MAC protocol. The model is a set of timed automata, and the analysis is performed using Uppaal [21]. The model for one node of the network is made of 5 timed automata. One of them is the *hardware clock* of the node. It is a one-state automaton, which produces *tick* events. The real time between two ticks is in an interval $[m, M]$, to model clock drift. The explicit `wait` instructions in the software are modeled by counting *ticks*. We will use a very similar structure. Our model in Lustre is *executable* and mainly used for simulations via the efficient compilation into C; but it is also formal, and could be used for formal verification.

3 Case Study

The case study is a MAC (Medium-Access-Control) protocol for wireless sensor networks (WSNs) called AreaCast [13], implemented on WSN430 nodes (see `www.senslab.info`). The radio component of a WSN430 node is the TI CC1100 [24]. For sake of simplicity, we represent three states only, instead of four: SLEEP, RX (receive), and TX (Transmit). The power-state model is given below.

When the radio is in state SLEEP, and command toRX occurs, it goes to state RX, but this will take S2RX units of time. This is represented by an intermediate (square-shaped) timed state with delay S2RX with a time-out transition (the arrow with a semicircle). The consumption per unit of time, in each of the modes and intermediate states, is denoted by γ. The values from the documentation are: γ(SLEEP)=2 mA; γ(RX)=15 mA; γ(TX)=16 mA; γ(S2TX)=γ(S2RX)=γ(RX2S)=γ(RX2TX)=γ(TX2RX)= γ(TX2S)=8 mA. The delays are: RX2S=TX2S=0.1 μs; S2RX=S2TX=88.4 μs; TX2RX=21.5 μs; RX2TX=9.6 μs.

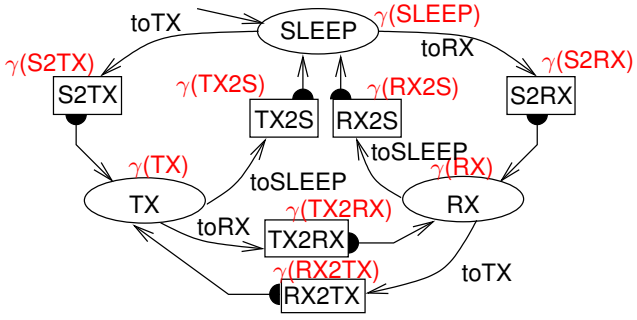


Figure 1: Power-State Automaton for the Radio

3.1 The AreaCast protocol

The protocol uses typical principles for sensor networks: nodes are put in sleep mode most of the time, and their cooperation is heavily based on *timing*.

When a node i wants to sends a message to a node j (as required by the routing level), and either the link $i \longrightarrow j$, or the node j, is faulty, some other nodes (say $k_1, k_2, ...$) that are situated in the same *area* as j can play the role of j, in a way that is transparent for i, the routing level, and the rest of the path to which the arc $i \longrightarrow j$ belongs. The protocol requires an estimation of the distance between nodes and each node has to be aware of its 2-hop neighborhood. See [13] for details, and Figure 6.

A normal transfer from i to j is made of four steps: i sends a RTS control packet (request-to-send), j sends back a CTS packet (clear-to-send), i sends the data packet, j sends an ACK packet (acknowledgment). When i initiates a message transfer to j with an RTS, and j does not answer correctly with a CTS, the nodes k are aware of the problem because they have received the RTS, and have seen no CTS from

j. Then, they have to decide together which k node will play the role of j. This is made possible by a ranking principle based on a distributed notion of distance, and requires no additional messages between the k nodes. Each k node *waits its turn*, according to its rank, which can be determined locally. If j does not answer during a delay Δ, the k node with the highest rank, say k_1, will decide to play its role; if k_1 is also faulty, k_2 will wait $2 \times \Delta$ until it decides to play the role of j; if k_1 and k_2 are faulty, k_3 will wait $3 \times \Delta$, etc. The number of nodes that can decide to replace j is 3.

3.2 Using the Existing WSNET Model

The execution platform is made of: the hardware of the nodes, plus the radio channel. A VP should allow to play with the protocol so as to obtain energy consumption evaluations, and to find functional and timing bugs.

In [13], the protocol is implemented in pure C, and the simulations performed with WSNET [9], without using the WSIM emulator (section 2.4 for details). Recall the WSNET simulator does not model clock drift. Moreover, a simulation with WSNET only is based on the Joule-per-bit consumption model which ignores overhearing (section 2.3). In [13], a trick is mentioned: to get a realistic model of energy consumption, all the mode changes of the radio are supposed to be explicit in the MAC protocol. This is not entirely satisfactory: the knowledge on the state of the radio cannot always be determined entirely by looking at the state of the MAC software. Some devices have spontaneous behaviors (e.g., loss of calibration, depending on time, not on explicit commands).

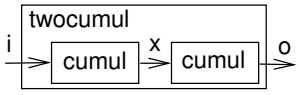
4 Synchronous Programming in a nutshell: Lustre

A Lustre program takes flows of inputs, produces flows of outputs, and has internal memory. The following program takes a flow of `int` values (e.g. `1, 2, 3, ...`), and outputs the flow of cumulated values (`1, 3, 6, ...`):

```
node cumul (i : int) returns (o: int) ;
let o = i -> pre (o) + i ; tel
```

The body of the program is a set of equations, defining the outputs and the internal variables. `o` is defined by: at the first instant (left-hand-side of the `->`), the value is that of `i`; then, forever (right-hand-side of `->`), the value is the previous one (`pre(o)`) plus the current value of the input `i`. The right part of equations can use: the `->` operator; all the ordinary operators (like `+`), interpreted point-wise; a conditional structure `if cond then e1 else e2`. A program can be called in another program, as a function. The following code describes the connection of two `cumul` programs:

```
node twocumul (i: int)
returns (o: int) ;
var x: int ;
let o = cumul (x) ;
    x = cumul (i) ;
tel
```



Ordinary Lustre programs are deterministic but we can make explicit calls to a `random` function implemented in C. The code below gives an example: the Lustre program imports the `alea` function as an *unsafe* function, i.e., a function that has side-effects. The Lustre program `loss` will be used to simulate losses on the links of the network. It calls the `alea` function to obtain a random number in the range $[0, 9]$.

```
// C implementation
static inline int alea(int i){ return (rand()%i); }
// Usage in a Lustre program
unsafe function alea(max: int) returns (a:int);
node loss(i: bool) returns (o: bool)
let   o = if alea(10) < 3 then false else i;    tel
```

5 Model of the Case-Study

The complete Lustre model, and the results of the experiments, are available on-line (omitted for blind review). For sake of simplicity, we model energy consumption for the radio only.

Fig. 2 shows the structure of the Lustre model. The Channel, Radio, CK constraints components constitute the hardware+physics virtual prototype. To play a given MAC protocol on this VP, we need a Lustre version of it, to be compiled with the other components in order to obtain the simulation code. This is obtained by encoding the automaton of Figure 6 into Lustre (section 5.1).

The model is made of: several nodes, and the radio channel. Since we do not represent the application level, we do not need a model of the physical environment of the sensors. Each node model is made of: the model of the radio (section 3) and the MAC automaton (Fig. 6). To abstract the routing and application levels, we expose a global input ToSend (*there is a message to send*) for each node. Each part of the model has an *explicit* notion of time (a `Ck` input). There is a global specification of the *constraints* between all these time notions (subprogram CK constraints).

The radio model in each node i outputs the energy consumed so far (e_i). The other wires are as follows: (2) the radio transmits messages on the channel; (3) the radio informs the MAC that the messages have been sent correctly; (4) the radio transmits the "channel is clear" information to the MAC; (5) the radio transmits to the MAC the messages it receives from the channel; (6) the MAC issues mode-change commands to the radio device; (7) the MAC gives the radio the messages to be transmitted on the channel; (8) the radio receives the messages from the channel; (1) the radio receives the "channel is clear" information.

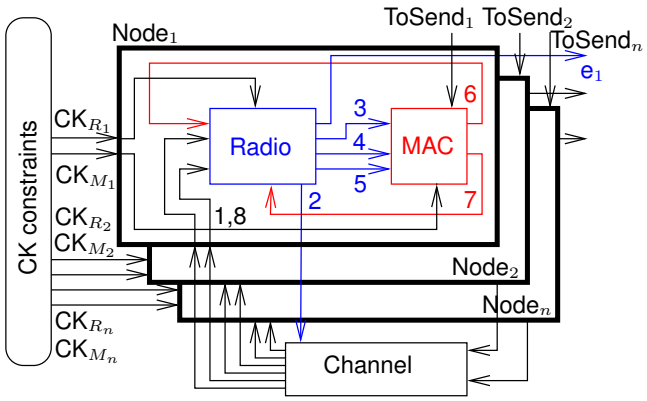


Figure 2: Structure of the Lustre Program

The block CK constraints of Fig. 2 works on the implicit base clock of the synchronous program, and produces each derived clock in the form of a sequence of Boolean values, with some degree of non-determinism. The code below generates one clock, according to the classical model of clock drift: the interval between two ticks is in $[n-p, n+p]$. It is exactly the discrete version of the clock generator described by a timed-graph in [23]:

```
node gen (n, p : int) returns (ck:  bool);
var  c, b : int;
let c  = 0 -> if pre(ck) then 0 else pre(c)+1;
    ck = (c=b) ;
    b  = n+alea(2*p+1)-p ->
         if pre(ck) then  n+alea(2*p+1)-p else pre(b);
tel
```

The various clocks of the model are the following: (i) the base clock of the Lustre program represents *real time*; it is used for the parts of the model that represent physical phenomena: the channel, and the radio model which outputs the energy consumed so far; (ii) each node i receives two clocks, one for the CPU noted CK_{M_i}, and one for the radio component (CK_{R_i}), for cases in which they are not the same.

5.1 Encoding Automata into Lustre

The Mealy machine of Fig. 3 illustrates the modeling needs: `A, C` are ordinary states; `B` is a "timed" state: when it is entered, the automaton will exit to `C` and emit `y` if `x` happens before 5 time units have occurred; otherwise, it will exit to `A` when the fifth time unit occurs, emitting `end`. Each state has an associated energy consumption per unit of (real) time, denoted by `eeA, eeB, eeC`.

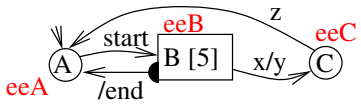


Figure 3: Example automaton with a timer and energy consumption

```
const eeA = 5 ; const eeB = 7 ; const eeC = 10 ;

node expaper (ck, start, x, z : bool)
returns      (end, y : bool ; -- outputs
              ee,sumee : int) ;     -- energy
var      A, B, C : bool ; -- states
         cpt : int;       -- timer for B
let
   A = true -> pre (A and not (start and ck)   or
                    C and (z and ck)           or
                    B and cpt=0 and ck);
   B = false -> pre (A and start and ck        or
                     B and not ( (x and ck)      or
                                 (cpt=0 and ck)));
   C = false -> pre (B and x and ck            or
                     C and not (z and ck)) ;
   cpt = 0 -> if pre (not B) then 5
              else if pre B and ck then pre cpt - 1
              else pre cpt ;
   end = B and cpt=0 and ck ;
   y   = B and x and ck ;
   ee  = if A then eeA else if B then eeB else eeC ;
   sumee = ee -> pre (sumee) + ee ;
tel.
```

Figure 4: Encoding Automata

Fig. 4 is the encoding into Lustre. The program is explicitly clocked by the input `ck`. The state variables `A, B, C` can change only when the clock input `ck` is true; the Mealy-style outputs `end, y` can be produced only when `ck` is true. Energy consumption, conversely, is counted on the base clock of the Lustre program, which represents real time.

The MAC protocol given in Fig. 6 can be encoded into Lustre following the same principle. The complete encoding is available on-line (reference omitted for the blind review). All the timed states representing explicit `wait` instructions of the MAC software are modeled by counting occurrences of the explicit clock of the node.

5.2 The MAC protocol

The automaton of Fig. 6 implements the full logic of the protocol, except the exchange of "Hello" packets and the distance estimation. It is made of two automata in parallel. The first one is a timer, which can be

started with input startA and generates the output endA when the timer expires. The main automaton is the algorithm itself. The variable me stands for the identifier of the node. There are alternating periods of *activity* and *sleep*. Typically, each node is sleeping during 5s, and then alive during 500ms. Before emitting a message, a node waits for a *random back-off* time, to avoid collisions (as in the WIFI 802.11 protocol).

State SLEEP is the sleeping period. When its timer expires, the automaton enters state 1, where it becomes sensitive to the arrival of a message to send from the routing level (toSend(dest)), or to the start of incoming radio packets (sRTS, sCTS, sDATA). It also starts the timer by issuing startA. In all other states, the occurrence of endA means that the period in which the node is alive has expired. If it is engaged in a message exchange (a RTS-CTS-DATA-ACK) sequence), it will finish it (endA is ignored on the path between states 2 and 6). Conversely, if the sender is in the state Random backoff, endA puts it to sleep, canceling the message transfer.

The path from 1 to 6 is the normal behavior of the sender: sending a RTS, waiting for the corresponding CTS, sending the data, waiting for the ACK. The path from 1 to 10 is the behavior of the node when it starts receiving a RTS which is for it (condition [me=D]); the path from 1 to 11 and then 13, 14 is the behavior of the node when it receives a RTS which is sent to a node to which it is an implicit relay. All other parts of the automaton deal with abnormal cases, e.g., a node x has sent a RTS to a node y, and received a CTS from y, but when it sends the DATA to y, y does not answer and, either an implicit node of y does the job, or the transfer has to be re-tried (maximum is Max=7).

5.3 Modeling the channel

The channel model "knows" that, if a node i has started sending a message at time t then some nodes $j_1, j_2, ..., j_n$ will start hearing it at time $t + \Delta$. We neglect Δ, but not the fact that transmitting a packet takes a time which is proportional to the size of the packet: the connection between a node and the channel is represented by two signals; for instance, receiving a RTS is split into sRTS (start) and eRTS (end). There is a probability p for a link to be faulty. Each time the channel has to distribute a message, and for each potential receiver `x`, it calls a `random` function to decide whether to distribute to `x` or not.

The geographical information is encoded into C, and we model only perfect channels, with the "unit disk" principle: two nodes that lie within distance $\leq D$ of each other hear each other. The "channel" component of Fig. 2 is a Lustre wrapper of this C implementation. The model is ready for a connection to more realistic channel models taken from, e.g., the WSNET library.

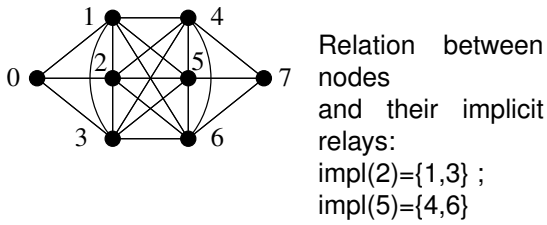


Figure 5: The Network

The connection graph (see example in Fig. 5) is encoded by a static matrix, which is used in the Channel model only. Maintaining the 2-hop neighborhood information and computing the fact that a node x is a potential implicit node of a node y, is supposed to be done, but not modeled here. We therefore encode the associated information in the C code of the channel, and make it available for the sotware.

6 Experiments

The goal is to show that our model of the software/hardware interface is sufficient to observe timing problems and energy consumption penalties. We use a small network of 8 nodes (see Fig. 5) with a perfect

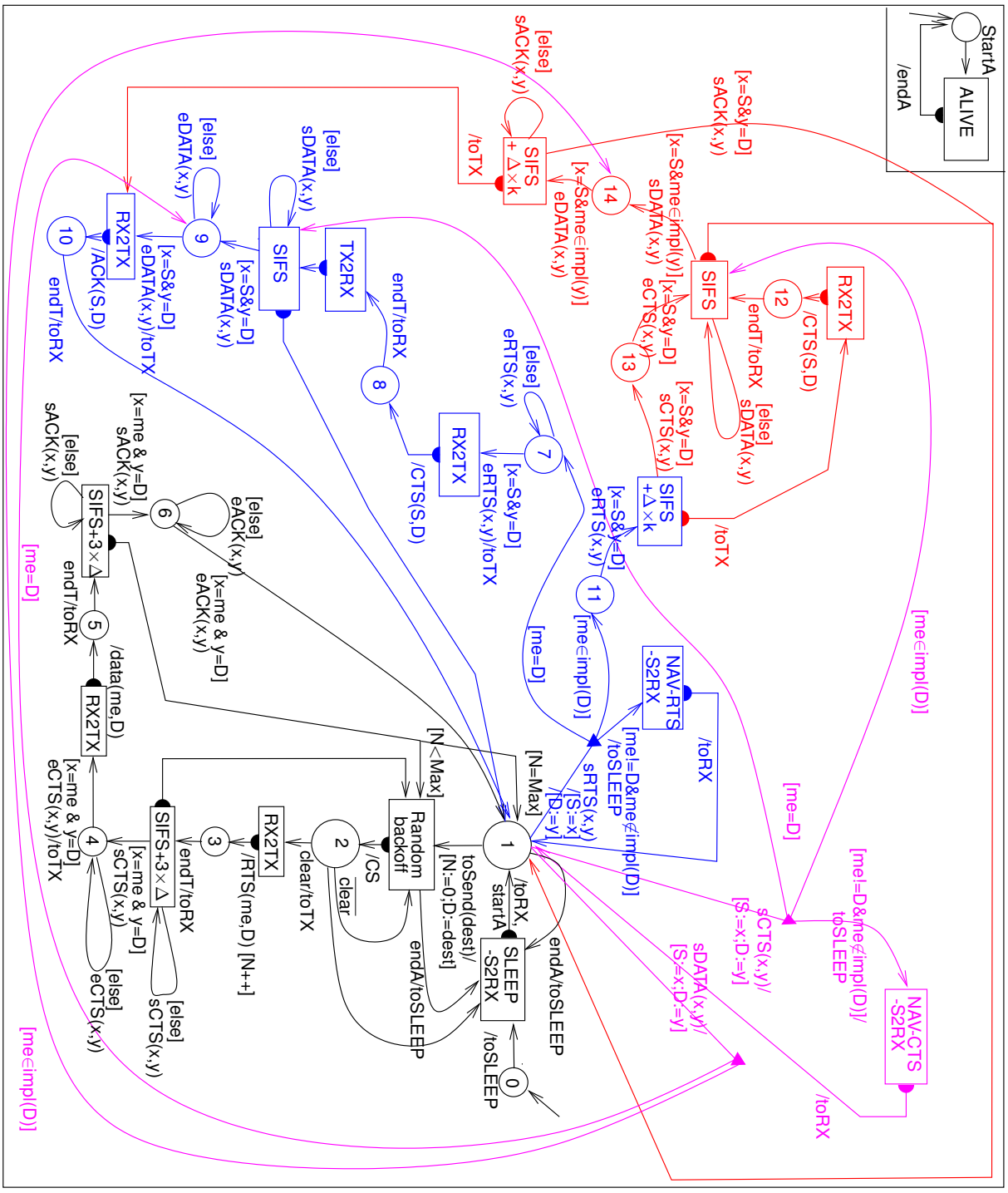


Figure 6: The Automaton for the MAC Protocol. SLEEP=5s; ALIVE=5ms; SIFS=10μs; DELTA = 25 μs; BACK-OFF=random([SIFS, SIFS + $2^{(N+3)}$])*20 μs; NAV_CTS = 161 μs; NAV_RTS = 174 μs; Max = 7. The syntax is that of Mealy machines with timed states and time-out transitions (as in section 3), augmented by *variables*; the inputs and outputs can have parameters (e.g., `sRTS(x,y)`); each transition can have a condition (e.g., [me=D]) and an assignment (e.g., S:=x). There are forked transitions: the trunk is labeled by some parametrized input associated with an assignment to local variables (e.g., sDATA(x,y)/[S:=x;D:=y], and the subsequent branches are labeled by conditions on the local variables (e.g., [me=D]).

channel. We always take the same clock for the radio and the MAC part of a node. The simulator is made of 1300 lines of Lustre, and 300 lines of manually-written C code for the channel.

6.1 Choosing the Granularity

The granularity G is the duration of one tick of the base clock. The constants of section 3 are approximated according to G. For $G = 10\mu s$: RX2S=TX2S=0 (instead of 0.1 μs); S2RX=S2TX=$9G$ (instead of 88.4 μs); TX2RX=$2G$ (instead of 21.5 μs); RX2TX=G (instead of 9.6 μs).

The granularity impacts the simulation speed. The time needed for executing one step of the Lustre program is 59 μs in our experiments; it is independent of G. If $G = 10\mu$s (resp. 100 νs), the time need to simulate $1s$ of real time is $5.92s$ (resp. $5963.77s$). Granularities less than 1 μs are not practicable for long simulations. However, they are not really necessary. A coarser granularity implies a coarser approximation of the constants, but the correctness of the protocol should not depend on precise relationships between those constants, otherwise it would not be robust. The approximation also impacts the evaluation of energy consumption, but this is the price to pay for non cycle-accurate models.

With the WSNET network simulator, for 50 nodes, without the details on the hardware/software interface, and with only one clock, simulating 40s of real time already takes 30 to 50s of machine time. To get a precise view of energy consumption, one needs to use a network of WSIM emulators, and the simulation speed is lower. With $G = 10\mu s$, and our detailed multi-clock model of 8 nodes, simulating 40s takes 240s of machine time. We think the cost is reasonable.

6.2 Observing the Effect of Faults

The model can be used to observe implicit nodes playing the role of nodes that can not be accessed due to faulty links.

In the following table, the path requested by the routing level is $0 \to 2 \to 5 \to 7$. The granularity is 10μs, all the clocks are equal to the base clock.

	p = 0 $0 \to 2 \to 5 \to 7$	p = 0.15 $0 \to 1 \overset{2}{\to} 6 \to 7$	p = 0.15 $0 \overset{2}{\to} 2 \to 5 \overset{5}{\to} 7$
e_0	2019	2274	5189
e_1	1831	2248	4436
e_2	1981	2332	4548
e_3	1831	2332	5452
e_4	1857	2145	4626
e_5	2007	2139	5136
e_6	1857	2081	4275
e_7	2032	2107	5227

When $p = 0$ (no faults), the path is indeed $0 \to 2 \to 5 \to 7$, with the energy consumption listed per node. When $p = 0.15$, various cases can occur. We show two of them. In the first one, the path is $0 \to 1 \to 6 \to 7$, and there were two tries for the link $1 \to 6$. In the second one, the path is $0 \to 2 \to 5 \to 7$, there were two tries for the link $0 \to 2$, and five for the link $5 \to 7$.

This kind of detailed result is needed for the evaluation of the trade-off implemented in the protocol (implicit nodes induce an over-cost in energy consumption, but since a message can take an alternative route, the number of retries and the delivery rate are better).

6.3 Playing with Time in the Protocol

Our model allows to observe the behavior of the protocol when two communicating nodes do not count time in the same way, because of clock drift. On the timing diagrams of Figs. 7, 8, 9, the dashed rectangles represent periods in which the software of a node executes an explicit `wait` instruction, counting on its own clock. The black rectangle represents the time it takes for the receiver to process the RTS message

and to prepare the CTS message. On Fig. 7, all nodes have the same deterministic clock (one tick every 10 ticks of the base clock). On Fig. 8 (resp. 9), the 1st implicit node (resp. the sender) has a faster clock (one tick every 5, resp. one tick every 3).

	fig. 7: $0 \rightarrow 2$	fig. 8: $0 \rightarrow 1$	fig. 9:lost
e_0	7937	7922	19822
e_1	8212	8738	19222
e_2	7934	7926	18804
e_3	8212	8197	19222

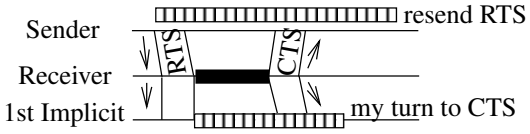


Figure 7: Ideal behavior of the RTS/CTS part

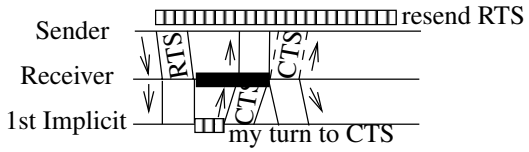


Figure 8: 1st implicit node faster than others

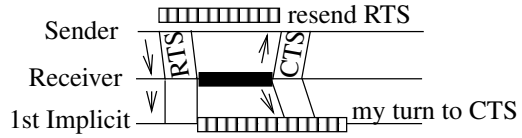


Figure 9: Sender faster than others

Experiments have been done for the requested path $0 \rightarrow 2$, the granularity being 10μs. On Fig. 8, the first implicit node plays the role of the legitimate node, because it does not wait long enough for its turn. It consumes a bit more, while the emitter consumes a bit less (it receives the CTS earlier). The others consume a bit less because the entire sequence RTS/CTS/DATA/ACK is shorter. On Fig. 9, the sender does not wait long enough to see the CTS: it re-emits the RTS 5 times, everybody is listening, hence an increase of energy. Even if the AreaCast protocol is somewhat resilient to the type of behaviors that occur when clocks get desynchronized, there is a penalty in energy consumption. Playing with clocks is compulsory to simulate this type of protocols that heavily rely on nodes counting time locally and trying to maintain a consistent view of the world around them. The kind of phenomena observed here cannot be observed with event-driven network simulators in which there is a single notion of time.

7 Conclusion

We have presented a generic structure for a virtual prototype of embedded systems, implemented in a synchronous language. The essential points are: (i) constraints between the various notions of time; (ii) an explicit description of the hardware/software interface with power-states models; (iii) the ability to take a

full description of the protocol into account (the same description by an automaton can be compiled for the simulation framework, or for the deployment on real sensors).

Our model is much more precise on time aspects than the models obtained with general-purpose network simulation tools, because we are able to observe what happens when the clocks differ; we can control how much they differ thanks to the constraint on clocks, to produce interesting cases. This is crucial for the type of protocols used in sensor networks, and is not provided in most network simulators.

We are also more precise on energy consumption than the usual network models without a full emulator of a dedicated platform, because we do model overhearing and devices with spontaneous behavior. The model is much simpler than a full emulator, which means that the observations are also easier to analyze when debugging. Finally, the simulation time is reasonable.

The next step is the implementation on the SensLab platform (see `www.senslab.info`), to compare the predictions of the model with actual measures.

References

[1] Omnet++. `www.omnetpp.org`. 1

[2] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced time automata. *Hybrid Systems: Computation and Control*, pages 147–161, 2001. 2.3

[3] L. Benini, R. Hodgson, and P. Siegel. System-level power estimation and optimization. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 173–178, New York, aug 10–12 1998. ACM Press. 2.3

[4] C. Bernardeschi, P. Masci, and H. Pfeifer. Early prototyping of wireless sensor network algorithms in PVS. In *Computer Safety, Reliability, and Security*, volume 5219 of *LNCS*, pages 346–359. 2008. 2.5

[5] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre, a declarative language for programming synchronous systems. In *14th POPL*, Jan. 1987. 1.2

[6] P. Caspi, C. Mazuet, and N. Reynaud-Parigot. About the design of distributed control systems: the quasi-synchronous approach. In *Proc. Safecomp'01*, volume 2187 of *LNCS*. Springer Verlag, Sept. 2001. 2.1

[7] J. Cornet, F. Maraninchi, and L. Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *Design Automation and Test in Europe (DATE)*, pages 9–14, Munich, Germany, Mar. 2008. 2.2

[8] J. Eriksson, F. Österlind, N. Finne, A. Dunkels, N. Tsiftes, and T. Voigt. Accurate network-scale power profiling for sensor network simulators. In *Wireless Sensor Networks*, volume 5432 of *Lecture Notes in Computer Science*, pages 312–326. 2009. 1

[9] A. Fraboulet, G. Chelius, and E. Fleury. Worldsens: development and prototyping tools for application specific wireless sensors networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 176–185, New York, NY, USA, 2007. ACM. 1, 2.4, 3.2

[10] A. Gamatié and T. Gautier. Synchronous modeling of avionics applications using the SIGNAL language. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 144–151. IEEE Computer Society, 2003. 2.1

[11] F. Ghenassia. *Transaction Level Modeling With SystemC: TLM Concepts And Applications for Embedded Systems*. Springer-Verlag, 2005. 1, 2.2

[12] W. R. Heinzelman, A. Ch, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *33rd Annual Hawaï International Conference on System Sciences*, pages 3005–3014, 2000. 2.3

[13] K. Heurtefeux, F. Maraninchi, and F. Valois. Areacast: a cross-layer approach for a communication by area in wireless sensor networks. In *17th IEEE International Conference on networks (ICON'11)*. IEEE, dec 2011. 3, 3.1, 3.2

[14] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *ACM Conference on Embedded Systems Software, EMSOFT 2007*, Salzburg, Austria, oct 2007. 2.1

[15] B. Kechar and L. Sekhri. Formal modelling and validation of a novel energy efficient cross-layer mac protocol in wireless multi hop sensor networks using time Petri nets. In *New Technologies, Mobility and Security, 2008. NTMS '08.*, pages 1 –5, Nov. 2008. 2.5

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM. 2.4

[17] F. Maraninchi, L. Samper, K. Baradon, and A. Vasseur. Lustre as a system modeling language: Lussensor, a case-study with sensor networks. In *SLA++P'07, ETAPS'07 Satellite Workshop on Model-driven High-level Programming of Embedded Systems*, Braga, Portugal, Mar. 2007. ENTCS. 1.2

[18] G. Merrett, N. White, N. Harris, and B. Al-Hashimi. Energy-aware simulation for wireless sensor networks. In *IEEE SECON'09*, Rome, Italy, June 2009. 2.4

[19] R. Milner. Communication and concurrency. In *International Series in Computer Science*. Prentice Hall, 1989. 2.1

[20] Open SystemC Initiative. *SystemC v2.1 Language Reference Manual (IEEE Std 1666-2005)*, 2005. `www.systemc.org`. 1, 2.2

[21] P. Pettersson and K. G. Larsen. UPPAAL2K. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 70:40–44, 2000. 2.5

[22] L. Samper, F. Maraninchi, L. Mounier, and L. Mandel. Glonemo: global and accurate formal models for the analysis of ad-hoc sensor networks. In *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, InterSense '06, New York, NY, USA, 2006. ACM. 1.2

[23] M. Schuts, F. Zhu, F. Heidarian, and F. Vaandrager. Modelling clock synchronization in the chess gMAC WSN protocol. In *Proc. of the Workshop on Quantitative Formal Methods: Theory and Applications (QFM'09)*, volume 13 of *Electronic Proceedings in Theoretical Computer Science*, 2009. 2.5, 5

[24] Texas Instruments. *CC1100, Low-Cost Low-Power Sub- 1 GHz RF Transceiver*, May 2009. 2.3, 3

[25] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press. 2.4