GLONEMO: Global and Accurate Formal Models for the Analysis of Ad-Hoc Sensor Networks

Ludovic Samper
France Telecom R&D
Email: Ludovic.Samper@francetelecom.com

Florence Maraninchi,
Laurent Mounier
VERIMAG
2, av. de Vignate, F38610
Email: Florence.Maraninchi@imag.fr,
Laurent.Mounier@imag.fr

Louis Mandel
VERIMAG
Email: Louis.Mandel@imag.fr

***Abstract*— We describe an approach for the formal modeling and analysis of ad-hoc sensor networks, at various levels of *abstraction*. It is *global* because it takes into account all the following aspects: a precise modeling of the hardware that implements a single node; the protocol layers; the application code; an abstract model of the physical environment as viewed by the sensors. The global model is *executable*, to enable validation by simulations, but we also aim at analyzing the global model with various formal validation tools (automatic test, runtime verification techniques, model-checking and abstract interpretations). Each technique or tool may need particular abstractions of the model. In this paper, we illustrate the whole approach with a simple model, and show what formal analysis can be performed on the model.**

I. INTRODUCTION

A. Ad-hoc Sensor Networks

Ad-hoc sensor networks have emerged recently, for a wide variety of application domains. A sensor network is a quite complex computer system. Ensuring a correct behavior of such a network is hard, and the better way to tackle the problem is to build *models* that can be simulated. Moreover, the power consumption is crucial. All the elements of a network have some influence on power consumption: the hardware of the nodes, the method used to access the radio functionalities, the communication protocols, the application, and even the environment of the network, that stimulates the sensors and is often the source of the main activity in the network. Once again, power consumption has to be estimated in advance, and this can be done by simulating a model.

B. Models and Simulation

There seems to be a wide agreement on the fact that traditional network simulators like NS [1] are not sufficient for ad-hoc sensor networks. In particular, they cannot be used to describe the hardware in an accurate way, which seems compulsory for power analyzes. A lot of approaches have been proposed for simulating ad-hoc sensor networks in a both accurate and efficient way. We compare these simulators to our proposal in Section VI below. None of these approaches is formalized; modeling a network is similar to a quite complex programming task, with a lot of threads. Libraries have been developed for some reusable elements of the models, like the protocols, but it is still hard to obtain an accurate and efficient simulator while preserving the faithfulness of the model. Moreover, those simulators do not help in modeling the *environment* of the network, i.e., the physical phenomena that have some influence on the sensors. Finally, as soon as the power analysis needs an accurate simulation of the hardware of a node, the problem becomes the same as simulating efficiently a large piece of hardware. Simulating 1000 nodes at the Register-Transfer-Level (RTL) is probably hopeless. People in the hardware design domain have tackled this problem by defining new levels of abstraction (like the so-called "*transaction-level modeling*" [2]) that are both accurate enough for a first approximate timing or power analysis, and fast to simulate. Developing such an approach for ad-hoc sensor networks requires a clear understanding of the abstractions than can be made on their behavior, while retaining their main power characteristics.

C. Contributions

We comment on the fact that there is no hope in obtaining models of sensor networks that are accurate, efficient, and that take into account all the aspects mentioned above, unless we are able to describe the elements of the model at various levels of abstraction. Moreover, such an understanding of the appropriate abstractions is hard to reach unless we develop *formal* models. For instance, having two formal models of a node at different levels of abstraction means we can try and prove that one is indeed an abstraction of the other, i.e., that the power estimations computed with the abstract model are always over-approximations of the accurate behavior.

We present an approach for the *formal* modeling of ad hoc sensor networks, with the following aims: 1) the modeling formalism has a clear operational semantics, independently of any execution engine; 2) we take all aspects into account in the same formalism: the hardware that implements a single node; the protocol layers; the application code; the physical environment as viewed by the sensors; 3) we show how a complete model can be build modularly, possibly with different levels of details; 4) we list existing analyzes and tools that can be applied to the formal model.

The formalism we choose is expressive enough so that we do not have to make a priori abstractions when modeling a sensor network. It is based on a clean and simple parallel construct, usable at two levels: the physical parallelism between the nodes of the network, and the physical or logical parallelism inside a node. Physical parallelism inside a node may be due to the presence of several pieces of hardware (sensors, CPU, dedicated piece of hardware for the MAC protocol, ...). Logical parallelism accounts for the presence of several processes, either executed on top of an operating system, or statically scheduled.

Our formalism is *executable*. Simulations are feasible for several thousands of nodes. Formal analyzes are possible, for two kinds of properties: 1) the consistency between the abstraction of a component used in the global model, and the precise description of the same component (for instance the hardware); 2) global safety properties of the network (*safety* [3] properties are the most interesting ones because they are preserved by abstractions).

The rest of the paper is organized as follows: Section II describes a toy example; Section III describes the formal model we use, and a language that implements it; Section IV uses this formalism to model the toy example; Section V lists the possible uses of the model; Section VI compares our approach to existing ones; Section VII concludes.

II. An Example

In a sensor network the nodes collaborate and exchange information in order to achieve a given service. We describe an example application, which protocols we choose, and why. The choices presented here are deliberately simple, but the formalism can be used to model more realistic cases.

We assume a *uniform* network, where most of the nodes are the same. However, to collect data, there can be one or several more powerful nodes called *sinks* that get the data and pilot the network. Sinks are not energy-constrained, hence we do not consider them here, since our model is intended to allow power analyzes.

A. Application

The goal of the network is to detect the presence of a radioactive cloud. Nodes have sensors to detect the radiations, and are scattered in a sensor field. When the cloud is detected by a node, it broadcasts alarm messages to the whole network.

B. Routing

In sensor networks communication patterns are specific, and the routing protocol depends directly on the application. Unicast is not a sensor network communication model, whereas flooding or broadcast can be useful. In this paper, we use the simplest routing mechanism, which is *flooding*: each data packet is sent to the whole network. We could have chosen a sink oriented pattern, and modeled a protocol like directed diffusion [4] where the communication pattern is a kind of converge-cast. Flooding is not an optimal routing mechanism for our application. Since a sensor network is quite dense, two nodes may sense the same stimulus at the same time, hence they will send the same messages. With flooding their common neighbors will receive duplicated messages. See section V-A on observing this phenomenon by simulations.

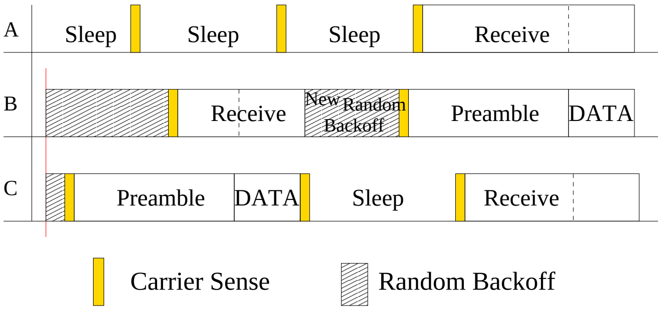

Fig. 1. Medium Access Control with back-off: B and C send messages. A and C are not in the range of each other.

C. Medium Access Control

To avoid energy waste, the MAC protocol is important. It plays a role in collision avoidance and collisions increase the energy consumption by involving re-transmissions. We implement a back-off. Figure 1 gives a possible timing behavior for three nodes A, B and C. The sender has to wait for a random time before emitting anything (when the routing is flooding, after a transmission all the neighbors of a node have immediately a packet to send, and the collision probability is high). Before an emission there is also a carrier sense: the sender checks whether the channel is free. If there is a signal on the channel, it delays the emission. One of the major sources of energy waste is idle listening. Idle listening is when the radio is listening whereas there is no signal on the channel. Hence, we implement in our model a MAC with preamble sampling. A preamble precedes each data packet for alerting the receiving node and all nodes in the network sample the medium with a common period.

This MAC protocol is representative of the protocols used in sensor networks. Our modeling formalism allows to describe such an algorithm, with random operations and timing.

D. Hardware

An accurate model of a network, allowing power analyzes, should include a model of the energy consumed by the hardware. In this paper, we use *abstract models* of the hardware consumption, not complete descriptions of the real hardware of a node. Power consumption can be divided into three domains: sensing, communication, and data processing. Energy expenditure in data processing and sensing is quite small compared to data communication [5]. Hence the first consumption model to include is the one of the radio. Furthermore, the switching times and thus the switching consumptions between the different power states cannot always be neglected. Hence, for a significant evaluation of the consumption, they must be included in the model.

III. A Modeling Formalism

A. Communicating Input/Output Interpreted Automata

Our modeling formalism is made of communicating parallel interpreted automata. Such an automaton has a finite number of explicit states, and its transitions are labeled by conditions on some *variables*, conditions on the presence or absence of some *signals*, assignements to the *variables*, and emissions of *signals*. The conditions may be combined freely with Boolean formulas. The signals whose presence is tested (resp. that are emitted) are called *inputs* (resp. *outputs*) of the automata. See, for instance, Figure 5.

Signals are used to synchronize automata, according to the *synchronous broadcast* communication mechanism. *Broadcast* means that if an automaton emits a signal, any number of other automata may be listening and reacting to it; *synchronous* means that the automaton thats sends a signal, and all the other automata that react to it do so "at the same time".

An interpreted automaton may have the expressive power of Turing machines, if the type of the variables is not constrained, making it possible to include algorithms in our models. The communication mechanism can be used to model pure synchronous systems, as well as asynchronous ones. If we need to include the VHDL description of the hardware of a node into our global model, it is possible; if we need to describe the asynchronous behavior of a communication protocol, it is also possible. All the automata are deterministic (although we may describe finite non-deterministic in the behaviors by introducing additional inputs called *oracles*). The parallel composition is also deterministic.

B. Quantitative Data

In order to model quantitative properties of our systems, we allow the states of the automata to be labeled by quantitative properties (energy consumed while in this state, time spent in this state, ...). These state labels do not interfere with the communication between the automata, because we cannot use their values in conditions. In the parallel composition of two automata, for each global state $(q1, q2)$, we gather the labels from $q1$ and $q2$. Depending on the quantity, the combination of the labels may be a sum, a max, etc.

If we want to attach a quantitative information to a transition between states $q1$ and $q2$, we simply add a transient state x between the two, and attach the information to this state.

C. An implementation in the functional style: ReactiveML

ReactiveML [6] is a functional-style language that extends ML [7] with reactive primitives. It can be used to program the automata of our modeling formalism, their compositions, and the state quantitative labels. Parallel composition is an easy-to-use primitive construct. Algorithms can be described by ML-like code. The synchronization between parallel processes is an implementation of the semantics described above. Any number of processes can write a value on the same signal, and all the processes that are listening to it see the set of values posted, and may combine them as they need. The value carried by a signal can be of any ML type.

The most natural way to execute a ReactiveML program is to exhibit the *basic clock* on which all the processes evolve. It is sometimes slow, for instance when we model multi-rate systems with very different rates. This is the case if we include in the same model: a piece of hardware that has to be observed each millisecond for the energy count to be accurate, and also a protocol that may be observed each second only; the basic clock would be 1 ms. See comments on simulation speed in section V-A.

One could implement an event-driven execution, to allow for better performances without changing the semantics of the model. However, as mentioned in the introduction, the bad performances of an accurate hardware simulation are not only due to the execution engine, but are often intrinsic to the level of details of the description. This is well known in the hardware design domain, and the only solution is to change the level of abstraction of the model itself.

D. Connection to Lucky

Since we do not have a complete knowledge of the physical phenomena involved in the environment, it is convenient to model the environment using a language based on *constraints*. We may express constraints between values of type real, at one moment in time, or relating successive instants of the behavior. For instance, we may express that the value of an input signal is within a given interval, and also that it increases, with a slope in a given interval. By expressing constraints that relate several values, we may describe quite complex behaviors.

The Lucky [8] execution engine is based on a constraint solver, for Boolean and numerical constraints. Executing a Lucky program produces a sequence of random values that respect the constraints. Lucky is connected to ReactiveML: a reactiveML program may include a process that corresponds to the execution of the Lucky engine for a given Lucky program.

Putting the model of the environment in our global model makes it *closed*. There is no need for inputs during the simulation. This methods allows to *program* realistic scenarios for the communications of our network. In particular, it allows to define correlated scenarios for all the inputs sensed by the nodes, and not only independent ones.

IV. Global Modeling

We describe a global model of the example explained in Section II, using the modeling formalisms of Section III.

A. Principles

Our global model is a set of communicating processes written in ReactiveML or Lucky. Figure 2 shows the processes and the information they exchange. The model is made of:

- a model of a node, expressing functional behavior and consumption properties; this model has one instance for each node, and all these instances are parallel processes;
- a model of the medium, i.e. the air in which the radio signals are propagated; this model "knows" the topology of the network, and includes the hypothesis we make on the radio link; we could include perturbations here; it receives signals from the MAC part of all nodes, and sends signals to the MAC parts of the appropriate nodes, w.r.t. the topology.

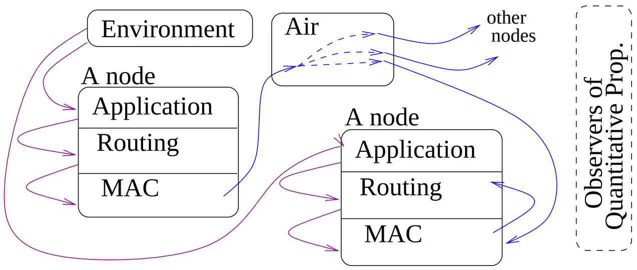

Fig. 2. Processes and Communications

```
let process send_alarm self cloud_pos =
if (present_cloud self cloud_pos) then
  if (not self.node_pre_present_cloud) then
    let new_packet = make_packet() in
      emit self.application_to_routing
           (new_packet);
      self.node_pre_present_cloud  <-
        present_cloud self cloud_pos;
```

Fig. 3. Part of the application code

- a set of *observers*, i.e., processes that do not interfere with the others, but look at their current states in order to compute a global consumption from all their labels.
- the model of the environment, written in Lucky and included as a particular process; it sends signals to the application part of each node.

The model of a node is further detailed. It is described as the parallel composition of: a process for the functional behavior of the application (the algorithm implemented on the node); a process for the functional behavior of the routing; a process for the functional behavior of the MAC protocol. All these models have quantitative properties attached, expressing the power consumed by the hardware in each state. The application process sends signals to the routing process, which sends signals to the MAC process, for emission. The MAC process can also sends signals to the routing process, in case of reception.

B. The Application

The application part of each node sends an alarm signal to the routing part when it receives a signal from the environment and determines that it is under the cloud (it compares its position with the cloud position). This alarm packet is sent only once for each detection of the cloud (the edge between “no cloud” and “cloud”). If the cloud does not move, then the nodes under the cloud will send only one alarm packet.

The application is included in the model as a detailed algorithm written as a ReactiveML process. The style of the code is illustrated by Figure 3.

C. The Environment

We model the moves of a cloud under the influence of the wind. Figure 4 is the Lucky program. The variables Wind_x and Wind_y represent a two-dimensional wind, which does not vary a lot. The cloud is a disk whose center has the coordinates x_cloud and y_cloud. The constraints may involve expressions like pre x_cloud, to talk about the previous value of the x_cloud variable. This allows to express constraints on sequences. The effect of the wind on the cloud is described by constraints of the form: if Wind_x >= 0.0 then ((x_cloud-pre x_cloud) <= Wind_x and (x_cloud-pre x_cloud) >= 0.0). Generating a sequence of values for the tuple (x_cloud, y_cloud) could give: (395.49, 385.98), (395.86, 386.15), (396.22, 386.33), ...

```
inputs {}
outputs {
 x_cloud: float init 350.0 max 700.0 min 0.0;
 y_cloud: float init 350.0 max 700.0 min 0.0;}
locals {
 Wind_x : float min -0.5 max 0.5 init 0.0;
 Wind_y : float min -0.5 max 0.5 init 0.0;}
nodes { init :stable;
        s_on   : stable;}
start_node { init }
transitions {
 init -> init cond
   if    (pre Wind_y - Wind_y) >= 0.0
   then  (pre Wind_y - Wind_y) <  0.05
   else  (pre Wind_y - Wind_y) > -0.05
  and
    abs ( pre Wind_x - Wind_x) < 0.05
  and
   (if  Wind_y >= 0.0
    then ((y_cloud-pre y_cloud) <= Wind_y
      and (y_cloud-pre y_cloud) >= 0.0)
    else ((y_cloud-pre y_cloud) <= 0.0
      and (y_cloud-pre y_cloud) >= Wind_y))
  and
   (if  Wind_x >= 0.0
    then ((x_cloud-pre x_cloud) <= Wind_x
      and (x_cloud-pre x_cloud) >= 0.0)
    else ((x_cloud-pre x_cloud) <= 0.0
      and (x_cloud-pre x_cloud) >= Wind_x))
}
```

Fig. 4. Lucky program for the environment

D. The Protocol Layers

1) Medium Access Control: The MAC protocol implemented in our global model is a preamble sampling MAC protocol (an example behavior was given on Figure 1). Figure 5 is the algorithm, described by an automaton. We give a simplified version, because the complete one would be two much detailed. In our model, the MAC protocol is a ReactiveML process that encodes the automaton, with all the details of the algorithm and the data structures exchanged. This process receives inputs from the routing part of the model (*packet to send*) and from the model of the medium (*channel busy*, *channel free* for instance). Additional signals (not shown here) are used to synchronize this automaton with the power consumption model (see IV-E below).

The effect of the MAC protocol can be observed by replacing the above protocol by another one (for instance the WiseMAC proposed by El Hoiydi et al [9], which adapts the length of the preamble to avoid sending a too long preamble before the data; or the MFP proposed by Bachir et al [10]; or a synchronized MAC protocol like S-MAC [11]).

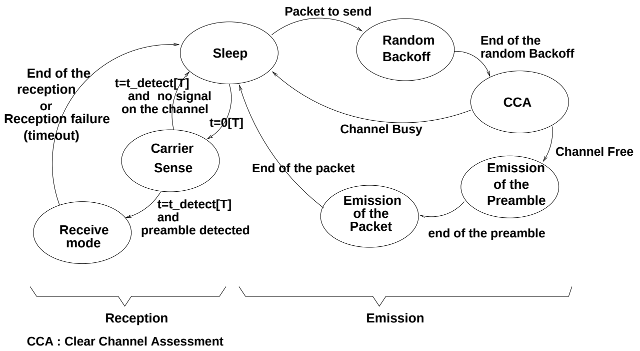

Fig. 5. Functional model of the MAC protocol

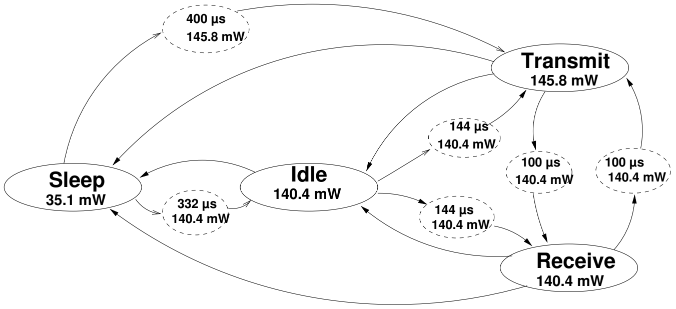

Fig. 6. Energy consumption of the radio

2) Routing: The routing we model is flooding: each node retransmits all the packets it receives except the ones it has already forwarded, hence preventing packets from doing loops in the network. The model is quite simple: the application part of the model of a node sends a packet to the MAC part, which sends it to the model of the air. The air then sends the packet to the appropriate set of nodes. The MAC part of the nodes receives the corresponding signal, and transmits it to the routing part of the model. This is where the check for loops is implemented. If the packet has not been seen yet, the routing part of the model sends a signal to the MAC part, which again sends it to the air. The routing is included in the model as a detailed algorithm written as a ReactiveML process.

E. The Hardware of a Node

The application and the protocols are included in our global model as simple but precise *algorithms*. The hardware could also be included as a precise description (e.g., in VHDL), but this would make the model quite complex, and this level of detail is not always needed. In this paper, we show how to define an abstract view of the hardware, concerning the energy consumption. This abstract model could be compared to a precise one (see section V).

1) The model: The consumption of the hardware is included in our model in the form of an automaton that describes the relevant states of the radio (see Figure 6). The actual numbers correspond to some consumptions that have been measured on a Freescale MC13192 [12]. The main states are `sleep`, `idle`, `transmit` and `receive`. The dashed-line states are transient states included in the model to represent the consumption of a transition between those main modes (see paragraph III-B).

In `sleep` mode, the radio is switched off and consumes nearly zero. In `idle` mode, the radio consumes as much as when it is receiving but there is nothing on the channel. In this mode, the radio is performing a carrier sense or is waiting for a signal on the channel. This state corresponds to Idle Listening. In `transmit` mode, the node is transmitting a signal on the channel. The corresponding consumption depends on the transmit power but we assume that the power of the transmission is constant for each node and that it is static. It would be very easy to describe different nodes each transmitting at a fix power. We could also imagine that a node can change dynamically its transmission range by increasing the transmit power which would consequently increase the consumption. To integer the power control in our tool we would have to admit several but a finite number of different transmission powers, each of which having a specific consumption. In `receive` mode, the node is receiving and decrypting a signal. The corresponding consumption is not far from the `transmitting` consumption. We assume that sending a preamble or a data packet is equivalent regarding the radio state and hence the consumption.

Our model uses additional signals that are not shown on the picture to ensure that this automaton is strongly synchronized with the automaton of the MAC algorithm (Figure 5). For instance, when the MAC automaton is in state `CCA`, the consumption model automaton is in state `Idle`.

2) Extension: CPU and memory consumption: Even if communicating consumes the most energy, the consumption of the microprocessor cannot be neglected. Yuan et al [13] assume it can take up to 30% of the total energy consumption.

If we want to model the energy consumption of the CPU, or that of the memory, we can do that with additional parallel automata, in the style of Figure 6, which means: 1) identifying relevant states for the consumption; 2) finding values to associate with these states, for instance with real measures; 3) relating such a consumption automaton to the other automata that describe the application and the protocols.

The difficult part is point 3. In the model above, it was easy to determine how the radio consumption is influenced by the behavior of the network, and we included this in the model: the radio consumption automaton (Figure 6) is driven by the MAC algorithm (Figure 5). But for the consumption of the CPU and the memory, it is intrinsically more complex.

Concerning the CPU, the consumption model could be *Dynamic Voltage Scaling (DVS)*. The model is an automaton that has one state per voltage level, and transitions that reflect what voltage level changes are possible in the hardware. The conditions on this automaton have to be related to the behavior of the software that runs on the CPU. A solution is to analyze the code first, statically, in order to insert DVS commands. In this case, the model is quite easy to write: a special instruction in the code itself determines the transitions in the consumption automaton. Other solutions may be based on dynamic measures of the CPU activity. In all cases, the intrinsic difficulty is in finding a DVS solution, not in modeling.

For the memory, the problem is roughly the same.

3) Extension: including an OS: If we want to include an operating system into the model, we will not include it as detailed algorithms. Instead, we will build an abstract model of it, identifying the relevant states that have some influence on the consumption. For instance, we could model the scheduler just to know whether a process is running. When there is no process running, the activity of the CPU is reduced, and so is its consumption.

V. Using the model

A. Simulations

First of all, since the model is executable, we can simulate it. We can observe the behavior in a graphical window: the cloud moves, triggers the activity of the nodes, we can observe the energy spent and the way messages travel. With 500 nodes, and a basic clock representing $10^{-4}s$, the simulation runs at a sufficient speed for the observations. 5000 nodes with a basic clock of $10^{-3}s$ is also feasible. Since the graphical display takes a large portion of the computation time, we can obtain even more efficient simulations by removing it. We should then include *observer* processes in parallel of the model, to compute relevant quantities and produce output files for drawing curves.

An observation that can be made on the model, thanks to the modeling of the environment, is that flooding is not a good choice for our application: two nodes may sense the same stimulus at the same time, and send the same messages. Their common neighbors then receive duplicated messages. If we had modeled the environment with Poisson laws to simulate the packet arrival at each node, two neighboring nodes would have been very unlikely to send the same message at the same time. Our environment model generates correlated signals for the sensors that are under the cloud, and is therefore realistic.

B. Formal validation

Now, since the model is formal, we can apply formal validation techniques and tools. Technically, it means extracting models in a form usable by the validation tools, from the ReactiveML code. This is not done yet, but since ReactiveML and Lucky have a formal semantics and a simple parallel construct, it is only a technical problem.

The first use we have in mind is the validation of the *abstractions* that are needed for the model to be of a reasonable complexity. For instance, we think that we should never include in the model a full description of the hardware, at the abstraction level that is needed for precise energy evaluations, i.e., the RTL level. But if we include an abstraction of it, we should *prove* that: 1) it is indeed an abstraction of the real hardware, and 2) the composition with the rest of the model preserves this abstraction. Point 1) should be tackled with automatic formal verification techniques like model-checking, to compare our abstract model with an RTL description of the CPU. Point 2) has to be proved manually on the semantics of parallel composition (easy).

The second use is the verification of *global* properties. There are three difficulties: the global properties related to the life span of the network are *quantitative* properties, for which there exist very few efficient verification tools; the global model includes software pieces, for which most interesting properties are undecidable; and a global model with 1000 nodes is likely to cause state explosion in any automatic verification tool. Our approach is based on abstractions. First, we need to abstract the quantitative properties into logical ones, by discretizing things: it is sufficient to define a small number of discrete energy states for each node, including the zero-energy one, and to talk about the global state in which a given proportion of the nodes have reached this zero-energy state. For the undecidability and size problems, the only solution is to find appropriate abstractions of the models, and to use *safety* properties that are preserved by abstractions (i.e., if an automatic tool declares a property true on the abstract model, it is true on the concrete one; the tool implements an approximate analysis, meaning it may also answer “don’t know”).

We are now studying the safety properties of the form: *after time T, the system still has more than x % of the nodes alive*. To be able to prove such a property automatically, we need to define abstractions of the software parts of the model.

VI. Related Work

NAB [14] is a network simulator written in ML, aimed at: scalability, visualization, and a clean, flexible architecture. The arguments for using ML instead of C are the same as ours, but ReactiveML is even better because parallelism is a primitive construct. NAB is not particularly well suited for an accurate modeling of the energy.

The nesC [15] language is a programming language, and does not help in modeling the hardware or the environment. However, it would be interesting to study the integration of nesC code into our model. The semantics of parallelism in nesC is not too far from the semantics of ReactiveML.

Avrora [16] 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 [17] 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.

PowerTOSSIM [18] makes interesting abstractions on the power used by a node, by relating it to the number of packets transmitted, the number of instructions executed, etc. This could be the basic of some abstract models of the hardware in our framework.

AEON [19] 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. In section IV we used a similar approach for the radio consumption model. AEON allows to observe the impact of the energy management primitives of TinyOS [20]. This is probably a good starting point if we want to include a model of the operating system in our framework.

Finally, 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 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.

VII. Conclusion

We demonstrated the use of a modeling formalism on an example sensor network. We showed how to include in the same model: precise algorithms for the application and the protocols, abstract consumption models for the hardware, and a non-deterministic model for the environment. We also explained how to extend this simple example if we need, for instance, a more detailed model of the hardware. We think that the main advantage of our modeling framework is the possibility to replace an abstract model of a component by a more precise one, and to prove that it is indeed an abstraction.

The simulation is feasible for hundreds of nodes. Moreover, since the model is formally defined as a set of parallel processes, formal validation techniques can be used.

We are working on the connection of the model to the automatic validation tools available at Verimag [21], [22].

References

[1] "The Network Simulator - ns-2." [Online]. Available: http://www.isi.edu/nsnam/ns/

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

[3] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977.

[4] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks." in *MOBICOM*, 2000, pp. 56–67.

[5] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey." *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.

[6] L. Mandel and M. Pouzet, "Reactiveml, a reactive extension to ml," in *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[7] X. Leroy, "The Objective Caml system release 3.09 Documentation and user's manual," INRIA, Tech. Rep., 2005.

[8] E. Jahier and P. Raymond, "The lucky language reference manual," Verimag Technical Report, Tech. Rep. TR-2004-6, 2005.

[9] C. C. Enz, A. El-Hoiydi, J.-D. Decotignie, and V. Peiris, "Wisenet: An ultralow-power wireless sensor network solution." *IEEE Computer*, vol. 37, no. 8, pp. 62–70, 2004.

[10] A. Bachir, D. Barthel, M. Heusse and A. Duda, "Micro-Frame Preamble MAC for Multihop Wireless Sensor Networks," *accepted, ICC*, 2006.

[11] W. Ye, J. S. Heidemann, and D. Estrin, "Medium access control with coordinated adaptive sleeping for wireless sensor networks." *IEEE/ACM Trans. Netw.*, vol. 12, no. 3, pp. 493–506, 2004.

[12] "Motorola mc13192 data sheet," Motorola freescale, 2005. [Online]. Available: freescale.com/files/rf_if/doc/data_sheet/MC13192DS.pdf

[13] Lin Yuan and Gang Qu, *Energy-Efficient Design of Distributed Sensor Networks*. CRC press, Oct. 2004, ch. 38.

[14] EPFL, "Network in A Box." [Online]. Available: http://nab.epfl.ch/

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The *nesC* language: A holistic approach to networked embedded systems," *j-SIGPLAN*, vol. 38, no. 5, pp. 1–11, May 2003.

[16] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable Sensor Network Simulation with Precise Timing," *Proceedings of IPSN*, 2005.

[17] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras, "ATEMU: A Fine-grained Sensor Network Simulator," *Secon*, 2004.

[18] V. Shnayder, M. Hempstead, B. rong Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications." in *SynSys*, 2004, pp. 188–200.

[19] S. G. Olaf Landsiedel, Klaus Wehrle, "Accurate prediction of power consumption in sensor networks," in *Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, Sydney, Australia, May 2005.

[20] TinyOS Team, "Tinyos." [Online]. Available: www.tinyos.net

[21] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE," *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, Sept. 1992.

[22] M. Bozga, S. Graf, and L. Mounier, "If-2.0: A validation environment for component-based real-time systems," in *Proceedings of CAV'02 (Copenhagen, Denmark)*, ser. LNCS, K. L. Ed Brinksma, Ed., vol. 2404. Springer-Verlag, July 2002, pp. 343–348.