Modeling of Sensor Networks Using XRM

Akim Demaille Sylvain Peyronnet Benoît Sigoure
EPITA Research and Development Laboratory (LRDE)
14-16 rue Voltaire
94276 Le Kremlin-Bicêtre
France
akim@lrde.epita.fr syp@lrde.epita.fr Benoit.Sigoure@lrde.epita.fr

***Abstract*—Wireless sensor networks are composed of small electronic devices that embed processors, sensors, batteries, memory and communication capabilities. One of the main goals in the design of such systems is the handling of the inherent complexity of the nodes, exarcerbated by the huge number of nodes in the network. For these reasons, it becomes very difficult to model and verify such systems. In this paper, we investigate the main characteristics of sensor nodes, discuss the use of a language derived from Reactive Modules for their modeling, and propose a language (and a tool set) that facilitate the modeling of this kind of system.**

I. Introduction

Wireless Sensor Networks (WSNs) are composed of very large numbers of small electronic devices that embed processors, sensors, batteries, memory and communication capabilities. Since such networks are conceptually very different from classical networks, new algorithms that can handle their specificities are required.

Several methods can be used to analyze the correctness and performance of such distributed algorithms. One of them is probabilistic model checking, an algorithmic method for the verification of probabilistic systems with respect to quantitative properties. The method is based on the construction of a mathematical model of the system and on the expression of the specification in some temporal language. The model represents all the possible states of the system, and the probabilities of the transitions that can occur between these states. More precisely, evaluating the satisfaction probability of a temporal property is reduced to the resolution of a system of linear equations over the state space. However, due to the state space explosion phenomenon during the modeling step, the representation of the transition matrix can be so large that the verification becomes intractable. To overcome this phenomenon, symbolic and numerical methods have been introduced in tools such as PRobabilistic symbolIc Model Checker (PRISM) [1]. Recently, a different model checking technique emerged implemented in Approximate Probabilistic Model Checker (APMC) [2]. Using this technique one can approximately compute the probability that a model meets its specifications. The computation time is not necessarily lowered, but the memory consumption drops considerably. These two approaches are complementary and a good way of using both methods is to verify a small but accurate system with a tool such as PRISM and then run APMC on bigger models but with less accuracy (APMC gives approximate results and does not handle nondeterminism).

Luckily, PRISM and APMC use the same input language: the PRISM input language, which is a variant of the Reactive Modules (RM) language for concurrent systems [3]. However, WSNs are so complex that it is mostly impossible for a human to write clear modeling of sensor nodes and their interactions. As a result it could be of interest to use pre-processing tools for modeling large and complex systems.

The main contribution of this paper is to present eXtended Reactive Modules (XRM), a syntactic pre-processor, particularly adapted for WSN, that can generate RM models. This tool set is available for download [4], and can be tested on line.

The structure of the paper is as follows. In Section II we present the specifics of WSNs that result in making the modeling of such systems difficult. In Section III we present eXtended Reactive Modules (XRM), a language suitable for the modeling of WSNs, and more generally very large systems. Finally, in Section IV we show, through an example, the interest of using XRM.

II. Wireless Sensor Networks

In this section we review the specificities of WSNs that result in making the modeling of such systems difficult, and present existing approaches.

A. Main characteristics of wireless sensor networks

More information can be found in [5], [6].

1) Network topology and scalability: The main issue with WSN is the size of the network (ranging from hundreds to millions of nodes) and the fact that the topology is complex. Indeed, sensor nodes are generally spread randomly, thus leading to a random communication topology. Moreover, due to the fact that nodes have a limited amount of energy, nodes can leave the network at any moment. This has a direct impact on the modeling: it is no longer possible to have the same behavior for each sensor node (it becomes dependent on the topology which is no longer uniform).

2) Fault tolerance: Distributed algorithms for WSNs must be fault tolerant. In order to verify such a property, it is mandatory to model failures. This is usually done by applying a Poisson distribution of not having a failure. It is not a problem in itself, but it adds transitions in the Continuous-Time Markov Chain (CTMC) underlying the system.

3) Environment: Sensors are supposed to be deployed directly into some extreme environment. Thus they may be modified directly by interactions with the environment: the modeling must take it into account.

4) Power consumption: Sensor node up-time is dependent on the battery lifetime. Power management is one of the main difficulties in the design of sensor nodes. From a modeling point of view, it is necessary to be able to model the power consumption. This adds again to the complexity of the modeling.

B. Existing Approaches to Model Generation

Approximate probabilistic model checking has already been applied to large systems such as WSNs. Because this paper focuses on an extension of RM, we will focus particularly on APMC.

In [**?**], in order to verify a WSN Media Access Control (MAC) layer, the authors wrote 306 lines of shell script and 287 lines of RM. The scripts generate C preprocessor (`cpp`) macros. These macros are run to model the WSN topology (a graph).

In an ongoing study of an atomic broadcast protocol [7], 389 lines of shell script driving `cpp` and 543 lines of RM are used. The use of a preprocessor was prompted, again, by the topology, but also by the lack of features such as lists.

In [8], as a simple approximation, the WSN topology is represented as a regular grid with holes. To handle the repetition, M4 [**?**], a powerful macro processor, is used. It is particularly well suited to extend language since it was designed and implemented to implement Rational Fortran (RATFOR) [9], an extension of Fortran. In order to generate various models, the authors drove M4 using a shell script. The whole implementation comprises almost 2,000 lines of M4, and 246 of script. In Section IV we investigate the suitability of XRM to generate similar models.

The common problems with these approaches are:

- total lack of verification, invalid code can easily be generated, eventually resulting in error messages about generated code;
- need to learn auxiliary tools;
- need to develop a whole framework, which may distract the practitioner from her true objective.

III. Extended Reactive Modules Languages

RM does not allow one to program large models, which penalizes tools based on it such as PRISM and APMC. To overcome these problems, one may extend either tool (and its language), in which case specific optimizations tailored to this tool are possible. Nevertheless we consider the limitation of the extension to a single tool as a shortcoming. In addition, working directly on production tools written in Java (PRISM and APMC) is inconvenient. So we are working toward designing an extended language, and its preprocessor.

eXtended Reactive Modules (XRM) provides a set of extensions that make RM suitable for WSN modeling. XRM addresses the issues raised in Section II-B: allowing the practitioners to focus on their modeling (instead of on the tools they need), to be more productive (by providing high-level easy-to-use constructs and informative error reporting), and to produce better modeling implementations (thanks to dedicated optimizations).

Optimizing is not just a feature, it is a requirement. Indeed, it is well known that factored implementations are less efficient than highly specialized hand-tuned implementations, a fact referred to as the "abstraction penalty". In model checking, where the model size can exceed the capacity of our tools, we cannot afford to pay the abstraction penalty: to be usable *in practice*, the generated models must be as well crafted as if it were hand-written.

A. Features

The XRM language is an extension to RM to ease the implementation of large scale models. In addition to syntactic details (e.g., one may write `end` instead of `endmodule`), it provides features to program large systems, in a structured manner. Specific optimizations are provided, including partial evaluation. Property specifications also benefit from these features.

The XRM preprocessor is implemented using the Stratego/XT tool set [10], a complete set of language-generic tools for program transformation. It features most prominently generalized parsing, capturing any context-free language, and the Stratego programming language, which provides powerful term-rewriting rules driven by strategies. This environment proved to be immensely useful to prototype and develop XRM.

We discuss the aspects of XRM that are of particular benefit when modeling WSNs. For a more thorough presentation of XRM and its implementation, see [11].

1) Scalability: The most troublesome drawback of RM for WSNs, and more generally for large modelings, is its lack of features to instantiate entities automatically. To work around that issue, most practitioners use auxiliary tools to generate a large number of modules from a *template*.

Although RM does provide means to instantiate a new module from another through variable renaming, this feature is rarely used for a number of reasons. First, the "generator" is an existing module that will be part of the resulting model, which prevents using pseudo variables names. As an example, consider a single chain of modules transmitting input from their "left" to their "right". One must write

```
module m1
  s1 : bool init false;
  [] s0 -> 1: s1 '= true;
endmodule
module m2 = module m1 [s0 = s1, s1 = s2];
module m3 = module m1 [s0 = s2, s1 = s3];
```

whereas it is much easier to use a template, a generator of modules:

```
module gen
  this : bool init false;
  [] prev -> 1: this '= true;
endmodule
module m1 = module gen [prev = s0, this = s1];
```

```
module m2 = module gen [prev = s1, this = s2];
module m3 = module gen [prev = s2, this = s3];
```

Unfortunately this implementation is incorrect in RM: `gen` will be part of the verification, it is a module, not a module template.

Private variables add complexity, since, they too, should be renamed. To address both issues of scalability and automatic generation, XRM features (multidimensional) arrays and loops. The above example can be written as

```
for i from 1 to 3 do
  module m[i]
    s[i] : bool init false;
    [] s[i-1] -> 1: s[i]' = true;
  end
end
```

2) Functions: Functions provide the most primitive form of factoring in programming languages. Because RM is not meant for large and repetitive models, it provides only a simple form of functions: non-recursive formulas. In XRM formulas may also contain "statements" such as updates:

```
formula tick = t' = t + 1;
```

RM formulas cannot have arguments, whereas XRM formulas accept any number of typed arguments. Besides expected types — `int`, `double`, and `bool` — XRM provides the `exp` type which enables passing arbitrary (well-formed) code. For instance

```
formula incr (exp var) = var' = var + 1;
```

XRM formulas can be recursive, a feature heavily used to specify large model properties (see Figure 5 for instance).

```
formula fact (int n) = n <= 1 ? 0 : n * fact (n - 1);
```

XRM provides additional built-in operators and functions, such as the generation of random values: `rand (42..51)` returns a different random integer at each invocation, while each occurrence of `static_rand (42..51)` will be substituted by a "constant" random integer.

Formulas, or rather functions, prove to be extremely handy to develop large models, see Figure 3 for a real case use of extended functions. Actually, none of these functions is a standard RM formula.

3) Specialization: In some cases modules represent similar components but cannot be obtained from each other by simply making name changes. In the previous chain example, consider the leftmost and rightmost modules. One may implement them specifically, but if the modules were actually on a grid there would be many more special cases. What if they were on a free form graph? XRM provides control structures that allow the user to specify points of variation in the implementation of modules, for instance

```
const int first = 1, last = 5;
for i from first to last do
  module m[i]
    s[i] : bool init false;
    if i = first then
      [] event -> 1: s[i] '= true;
    else if i = last then
      [] s[i-1]  -> 1: alarm '= true;
    else // i != last
      [] s[i-1]  -> 1: s[i] '= true;
    end
  end
end
```

or better (also valid when `first = last`)

```
module m[i]
  s[i] : bool init false;
  if i != last
    [] i = first ? event : s[i-1] -> 1: s[i] '= true;
  else // i = last
    [] i = first ? event : s[i-1] -> 1: alarm '= true;
  end
end
```

Note that in either case the code expansion requires intelligent evaluation of code. Consider an extreme case where `first = last = 1`, the expansion of the loop gives

```
module m[1]
  s[1] : bool init false;
  [] i = first ? event : s[1-1] -> 1: alarm '= true;
end
```

where the underlined expression must not be computed: it results in an out-of-bound access. In this case, the XRM compiler will properly generate three classes of different modules: leftmost, center, and rightmost. As a full scale use of this feature, see the formula `ears` in Figure 3: it has $2^4 = 16$ possible expansions, representing all the possible neighborhoods for a cell in a grid (including the extreme cases where a dimension is reduced to one or zero cells).

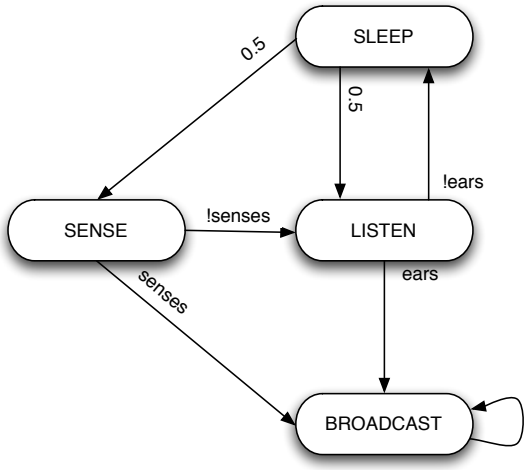
To account for less regular topologies while keeping the model as simple as possible, there are no constraints on "array" indexes. For instance you may declare and use only `s[i][i]` with `i` $\in [1..100]$ (100 slots). In most other programming languages, this requires declaring 100×100 slots.

4) Properties: Not surprisingly, every feature useful to model a large system helps to write its specifications. It was also noted in existing approaches (Section II-B) that the properties were also built using not only the same parameters, but also common formulas. Taking this common use into account, XRM features `properties ... end` sections into which the user can write its specifications (see Figure 5). The XRM compiler produces a separate file.

IV. Case Study

In [8] APMC is used to verify WSNs. The authors reported APMC suits WSN verification, although it exhibits a few shortcomings. Since then, these limitations have been addressed, most notably the addition of support for CTMC [12]. Overcoming the limitations of the input language, RM, prompted the development of XRM.

As a case study of XRM, we consider the re-implementation of the simple WSN verification proposed in [8]. For instance, consider a forest gridded by sensors whose purpose is to carry fire alerts to the edge. The modeling is presented in the following section, then its implementation in XRM, and finally a comparison with the ad-hoc approach from [8].



First the sensor is sensing its environment (state `sense`). If it detects fire, it goes directly to the state `broadcast` and stays in it forever (that is, it sends without interruption a signal to its neighbors). Otherwise it goes to state `listen`, where the sensor is listening to its neighborhood. If it catches a broadcast, it goes to `broadcast` in order to forward the information, otherwise it sleeps (state `sleep`) for a while: with equiprobability it goes back to `sense` or `listen`.

Fig. 1. Sensor node behavior taken from [8]

A. A Modeling

The topology of the WSN is modeled as a rectangular grid. If a sensor detects an event in its cell, it broadcasts the information to its four neighbors (see Figure 1). This simplistic definition of the neighboring of each sensor provides a realistic approach to wireless connectivity. Any sensor in any cell could be down, thus this grid topology is only used by our simulation mechanism to emulate a realistic geographic distribution. Related work on WSNs makes the same assumption, and these logical grids may be built using distributed algorithms [13], [14], or physically realized using a GPS and an election algorithm to ensure that at most one sensor is present in each logical cell of the grid.

With this grid model, it is semantically equivalent to model the sensor or the cell, but the latter is much easier. Each cell can be active (there is a live sensor) or inactive (there is none or it ran out of energy). Each cell has a simple behavior (Figure 1).

Real field experiments have demonstrated that wireless broadcast communication from small and low-cost units are usually unreliable. Often, messages fail to reach direct neighbors, and may communicate with more remote components of the field. This unreliable communication media is modeled here in the `LISTEN` state. Since each sensor independently may not be in `LISTEN` state when a neighbor broadcasts, a message may be lost. The probability law of the `LISTEN` state encompasses the algorithm's behavior and the medium losses.

To model power consumption, a simple model was chosen: each time a sensor leaves a state, a state-dependent amount of energy is subtracted. Now that CTMCs are supported by APMC, it is possible to relate consumption to time.

```
// Grid (even) dimensions.
const int X = 10, Y = 10;

// Initial energy, percentage of lost cells.
const int POWER = 15, LOSS = 50;

// States.
const int OFF = 0, SLEEP = 1, SENSE = 2,
          LISTEN = 3, BROADCAST = 4;

// Energy consumption for each state.
const int COST_SLEEP = 1, COST_SENSE = 1,
          COST_LISTEN = 3, COST_BROADCAST = 3;
```

Fig. 2. Parameters of the modeling

```
// Whether (x, y) are valid sensor coordinates.
formula valid (int x, int y) =
  0 <= x & x < X & 0 <= y & y < Y;

// Whether (x, y) is valid, and broadcasting.
formula broadcasts (int x, int y) =
  valid (x, y) & s[x][y] = BROADCASTS;

// Whether a neighbor of (x, y) is broadcasting.
formula ears (int x, int y) =
    broadcasts (x - 1, y) | broadcasts (x + 1, y)
  | broadcasts (x, y - 1) | broadcasts (x, y + 1);

// Whether the event (middle cell) is detected.
formula senses (int x, int y) =
  x = X / 2 & y = Y / 2;

// Consume c units of power.
formula consume (int x, int y, int c) =
  b[x][y]' = b[x][y] < c ? 0 : b[x][y] - c;

// Reach state st if energy allows it.
formula set_state (int x, int y, int st) =
  s[x][y]' = (0 <= b[x][y]) ? st : OFF;

module timer
  t : [0..666] init 0;
end

// Transition to the next state st, consuming c.
formula transition (int x, int y, int st, int c) =
  t' = t+1 & consume(x, y, c) & set_state(x, y, st);
```

The module `timer` and the formula `transition` maintain a global variable `t` which is the time. It is not used in the modeling, but in the properties to be verified.

Fig. 3. Formulas

The initial instant coincides with the detection of the fire, that is to say, the (center) grid cell broadcasts immediately. The robustness of the WSN is evaluated by measuring the probability that the alert is reported at specified time intervals. To test the robustness, we change the initial conditions: a percentage of sensors randomly lost on the grid.

B. The Implementation in XRM

XRM makes the implementation simple and concise: the whole implementation is included. Parameters are introduced

```
for x from 0 to X - 1 do
  for y from 0 to Y - 1 do
    module sensor[x][y]

      s[x][y] : [0..4] init (static_rand (0, 100) < LOSS) ? OFF : SENSE;
      b[x][y] : [0..POWER] init POWER;

      [] s[x][y] = SENSE     -> 1: transition (x, y, senses (x, y) ? BROADCAST : LISTEN, COST_SENSE);
      [] s[x][y] = LISTEN    -> 1: transition (x, y, ears (x, y) ? BROADCAST : SLEEP, COST_LISTEN);
      [] s[x][y] = SLEEP   -> 0.5: transition (x, y, SENSE, COST_SLEEP)
                            + 0.5: transition (x, y, LISTEN, COST_LISTEN);
      [] s[x][y] = BROADCAST -> 1: transition (x, y, BROADCAST, COST_BROADCAST);
    end
  end
end
```

Fig. 4. The WSN Model in XRM

```
// A broadcast in (x, y..Y - 1)?
formula X_broadcasts (int x, int y) =
  valid (x, y)
  & (broadcasts (x, y) | X_broadcasts (x, y + 1));

// A broadcast in (x..X - 1, y)?
formula Y_broadcasts (int x, int y) =
  valid (x, y)
  & (broadcasts (x, y) | Y_broadcasts (x + 1, y));

// A broadcast in the perimeter?
formula boundary_broadcasts =
    X_broadcasts (0, 0) | X_broadcasts (X - 1, 0)
  | Y_broadcasts (0, 0) | Y_broadcasts (0, Y - 1);

properties
  for T from 0 to 1200 step 100 do
    // Alarm triggered before instant T?
    P =? [ true U (t <= T & boundary_broadcasts) ];
  end
end
```

The properties to verify can be specified in a separated file, or embedded in the "master" XRM file, as is the case above, in order to share parameters and formulas.

Fig. 5. The WSN Properties in XRM

in Figure 2, useful formulas in Figure 3, and the grid itself in Figure 4. The grid is implemented via two nested `for` loops. Each sensor has two variables to store its current state and energy level. Then the set of possible transitions with specified probabilities are given.

The last section of our XRM file, containing the properties, is given in Figure 5. Specifying properties of the model to verify takes a similar amount of work for large systems. In the running example, the system meets its specifications if the alarm is properly delivered to the boundaries of the grid. Again, looping constructs are required, but given that side effects are prohibited, we use XRM's support of recursion to implement formulas such as `X_broadcasts`, which checks whether a cell in a column is broadcasting.

We can use APMC to approximate the probability that *several* formulas are verified in one run. In [8] this feature was heavily used to extract "delivery" graphs: the probability that the message was delivered at regular intervals of time. As a result, the properties need XRM features: functions (extended formulas), loops, and so on. Because properties can be embedded in the same XRM file as the model, they easily share the parameters and formulas.

C. Benefits of XRM

Using XRM vastly simplified the redevelopment of the model described by [8]. The initial implementation represents 246 lines of shell script, 1,769 lines of M4 general macros and 257 lines of dedicated macros. It takes about 10 lines of shell and 54 lines of XRM.

1) Domain Specific Compiler: The first and foremost advantage of using the XRM compiler instead of hand-crafted tools is almost invisible: because you actually use a true compiler rather than a simple macro processor, there are genuine validity tests and error reporting features. Developing a large model is tedious and error-prone, one may easily make out-of-bounds mistakes, swap arguments, etc. If lucky, the resulting code will be incorrect and rejected by the RM engine; yet the error will be reported against unreadable, non-indented, *generated* code instead of against the high-level source. If unlucky, the error will remain unnoticed, and will "only" render the verification meaningless. Using the XRM tool set, several high level sanity checks are performed (bounds checking, type checking, warnings for unreachable states, etc.) and possible errors are reported against the genuine source.

Ironically, although there is less point in looking at expanded XRM, it is by far more readable than that of hand-made preprocessors. Contrary to the latter ones that expand macros blindly, unaware of the structure of the host language, the former builds an Abstract Syntax Tree (AST) of the program which is transformed and pretty-printed back into properly indented RM source.

2) Domain Specific Syntax: Because XRM is truly developed as an extension to RM, its syntax was crafted to fit, unlike general purpose macro preprocessors. In addition, external tools require special tags separating macro language from host language; for instance, consider all the additional square brackets when using M4 in Figure 6. In addition, to avoid identifiers from one world to be captured by the other,

```
# Build the sensor at X,Y.
m4_define([rm_sensor],
[module S[]rm_id([$1], [$2])

  rm_state_definition([$1], [$2])
  rm_battery_definition([$1], [$2])

  rm_rule1([$1], [$2], SENSE,     m4_if(m4_eval(rm_event == rm_location([$1], [$2])), 1, BROADCAST, LISTEN))
  rm_rule1([$1], [$2], LISTEN,    rm_neighbors_broadcast($1, $2) ? BROADCAST : SLEEP)
  rm_rule2([$1], [$2], SLEEP,     0.5, SENSE,   0.5, LISTEN)
  rm_rule1([$1], [$2], BROADCAST, BROADCAST)

endmodule
])

m4_for([X], [0], rm_max_x, [1],
       [m4_for([Y], [0], rm_max_y, [1],
               [rm_sensor([X], [Y])
])])
```

This is the original code from [8]. Interestingly in their Fig. 2 the authors did not show exactly this code (entangled with M4 code), but rather some abstraction of the code they "meant". Contrast this to Figure 4.

Fig. 6. The WSN Model Generated in RM using the M4 Macro Processor

name conventions are required, hence all the `rm_` and `m4_` prefixes. XRM features scopes, that is to say that an identifier might serve different purposes in a different location without risking unexpected captures. Finally, the calling convention for functions can be tailored to be consistent with the host syntax: see how XRM functions support both *named* and *typed* arguments (Figure 3) as opposed to M4 for instance (macro `rm_sensor` in Figure 6).

3) Domain Specific Tool: Designed to fulfill our needs to model WSNs, and set to meet the expectations of the practitioner, XRM allows its users to be more productive. Its syntax, simple and consistent with RM, makes the learning curve gradual. Thanks to its dedicated tool set, the edit-compile-debug cycle is short. Because the tools exist, the practitioner is not distracted by the need to implement sophisticated shell-scripts. Finally, because the compiler features various optimizations (like dead code removal), the model might be even easier to verify.

V. Conclusion

In this paper, we have presented eXtended Reactive Modules (XRM), a syntactic pre-processor for Reactive Modules (RM) well suited for modeling large systems such as wireless sensor networks. Using XRM it is possible to generate RM models (suitable for PRISM [1] and APMC [2]). In the future, we might directly use the high-level model: highly factored, it is inexpensive to run directly in an environment such as APMC.

Acknowledgments

We thank Martin Bravenboer, from the Stratego/XT team, for his continuous help during the development of XRM. The anonymous referees suggested very useful changes in this paper. Stephen Frank reported mistakes.

References

[1] M. Z. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic symbolic model checking with PRISM: A hybrid approach," in *Tools and Algorithms for Construction and Analysis of Systems*, 2002, pp. 52–66.

[2] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, "Approximate probabilistic model checking." in *VMCAI*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 73–84.

[3] R. Alur and T. A. Henzinger, "Reactive modules." in *LICS*, 1996, pp. 207–218.

[4] "Xrm web page," 2006, http://projects.lrde.epita.fr/Xrm.

[5] D. E. Culler, D. Estrin, and M. B. Srivastava, "Guest editors' introduction: Overview of sensor networks." *IEEE Computer*, vol. 37, no. 8, pp. 41–49, 2004.

[6] 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.

[7] M. Cadilhac and J. Oudinet, "Private communication," 2006.

[8] A. Demaille, S. Peyronnet, and T. Hérault, "Probabilistic verification of sensor networks," in *Proceedings of the Fourth IEEE International Conference on Computer Sciences, Research, Innovation and Vision for the Future (RIVF'06)*, Ho Chi Minh City, Vietnam, February 2006.

[9] B. W. Kernighan, "RATFOR – a rational fortran," in *Workshop on Fortran Preprocessors, Pasadena, Calif.*, November 1974, p. 3.

[10] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9," in *Domain-Specific Program Generation*, ser. Lecture Notes in Computer Science, C. Lengauer *et al.*, Eds. Spinger-Verlag, June 2004, vol. 3016, pp. 216–238. [Online]. Available: http://www.cs.uu.nl/research/techreps/UU-CS-2004-011.html

[11] B. Sigoure, "eXtended Reactive Modules," EPITA Research and Development Laboratory (LRDE), Tech. Rep., 2006. [Online]. Available: http://publications.lrde.epita.fr/200607-Seminar-Sigoure

[12] T. Hérault, R. Lassaigne, and S. Peyronnet, "APMC 3.0: Approximate verification of discrete and continuous time markov chains," in *Proceedings of Qest 2006*, 2006, to appear.

[13] R. Friedman and G. Korland, "Timed grid routing (TIGR) bites off energy," in *Proceedings of MobiHoc 2005*, 2005.

[14] J. Hightower and G. Borriello, "Location systems for ubiquitous computing," *IEEE Computer*, vol. 34, no. 8, pp. 57–66, August 2001.