

Formal Modeling and Analysis of the OGDC Wireless Sensor Network Algorithm in Real-Time Maude

Peter Csaba Ölveczky and Stian Thorvaldsen

Department of Informatics, University of Oslo
peterol@ifi.uio.no stianth@ifi.uio.no

Abstract. This paper describes the application of Real-Time Maude to the formal specification, simulation, and further formal analysis of the sophisticated state-of-the-art OGDC wireless sensor network algorithm. Wireless sensor networks in general, and the OGDC algorithm in particular, pose many challenges to their formal specification and analysis, including novel communication forms, treatment of geographic areas, time-dependent and probabilistic features, and the need to analyze both correctness and performance. Real-Time Maude extends the rewriting logic tool Maude to support formal specification and analysis of object-based real-time systems. This paper explains how we formally specified OGDC in Real-Time Maude, how we could simulate our specification to perform all the analyses done by the algorithm developers using the network simulation tool ns-2, and how we could perform further formal analyses which are beyond the capabilities of simulation tools. A remarkable result is that our Real-Time Maude simulations seem to provide a much more accurate estimate of the performance of OGDC than the ns-2 simulations. To the best of our knowledge, this is the first time a formal tool has been applied to an advanced wireless sensor network algorithm.

1 Introduction

This paper describes the application of Real-Time Maude [17, 15] to the formal specification, simulation, and further formal analysis of the state-of-the-art *optimal geographical density control* (OGDC) wireless sensor network algorithm [22]. To the best of our knowledge, this work represents the first formal modeling and analysis effort of such a complex wireless sensor network system.

A wireless sensor network (WSN) consists of many small, cheap, and low-power sensor nodes that use wireless technology (usually radio) to communicate with each other [2]. Given the increasing sophistication of WSN algorithms—and the difficulty of modifying an algorithm once the sensor network is deployed—there is a clear need to use formal methods to validate system *performance* and *functionality* prior to implementing such algorithms.

In [19] we advocate the use of the language and tool Real-Time Maude [15, 17], which extends the rewriting logic-based Maude [3] tool to real-time systems, to formally specify, simulate, and further analyze WSN algorithms. The

Real-Time Maude specification language emphasizes expressiveness and ease of specification. The data types of a system are defined by *equational specifications*. Instantaneous transitions are defined by *rewrite rules*, and time elapse is defined by *“tick” rewrite rules*. Real-Time Maude supports the specification of distributed *object-oriented* systems, which is ideal for modeling a network system. The high-performance Real-Time Maude tool provides a range of analysis techniques, including: timed rewriting for simulation purposes; timed search for reachability analysis; and time-bounded linear temporal logic model checking. Real-Time Maude has been used to model and analyze a set of advanced real-time systems, such as large communication protocols [18, 8] and scheduling algorithms [13]. Such analysis has found subtle design errors not uncovered during traditional simulation and testing. We argue in [19] that Real-Time Maude’s expressive specification formalism, and the ease with which new forms of communication can be defined, should make it ideal to model WSN systems.

Jennifer Hou suggested to us her OGDC algorithm [22] for WSNs as a challenging modeling and analysis task. OGDC is a sophisticated state-of-the-art algorithm that tries to maintain complete sensing coverage of an area for as long as possible by switching nodes on and off. It has been simulated by the algorithm developers Zhang and Hou using the simulation tool ns-2 [12, 4].

The OGDC algorithm is an advanced algorithm whose formal specification, simulation, and analysis pose a set of challenges, including:

1. Modeling—and computing with—spatial entities such as coverage areas, angles, and distances.
2. Modeling broadcast communication with transmission delays and limited transmission range.
3. Modeling time-dependent behavior, such as use of timers, transmission delays, and power consumption.
4. Modeling *probabilistic* behaviors. For example, sensor nodes volunteer to start with certain probabilities, and different values are supposed to be “random values, drawn from a uniform distribution.”
5. Simulating and analyzing systems with hundreds of sensor nodes.
6. Analyzing both correctness and, in particular, performance.

This is indeed a challenging set of modeling and analysis tasks. This paper shows how Real-Time Maude met these challenges. In particular, during simulations of the algorithm, we are able to do in Real-Time Maude *all* the performance analyses that Zhang and Hou performed using the wireless extension of the network simulation tool ns-2 [12]. In addition, we have subjected the algorithm to *time-bounded* reachability analysis and temporal logic model checking.

By modeling transmission delays (which play a significant role in the definition of the OGDC algorithm), and by comparing our performance measures with the ns-2 simulation results, we found a discrepancy which *could* be explained by a (minor) weakness in the algorithm *if* the ns-2 simulations did not take the transmission delays into account.¹ To test this hypothesis, we also per-

¹ We have not received information of whether the ns-2 simulations actually took the transmission delays into account, only that it is likely that they did not.

formed Real-Time Maude simulations *without* considering transmission delays. The results of these simulations are quite similar to the ns-2 simulations. It is therefore tempting to conjecture that our original simulations provide a much more accurate estimate of the performance of OGDC than the ns-2 simulations.

Related work. Our work represents—to the best of our knowledge—the first formal modeling and analysis of such a sophisticated WSN algorithm as OGDC. Some attempts at using formal methods on WSNs have focused on modeling TinyOS using automaton-based formalisms (see, e.g., [5]), or have considered simple diffusion protocols for discovering routing trees [11]. Our paper [19] explains related work in more detail. That paper also suggests that Real-Time Maude might be a good candidate for formally modeling WSNs, and shows how certain features of such networks, including locations, distances, and communication can be easily modeled in Real-Time Maude. In contrast, this paper focuses on the OGDC case study: It shows how the general techniques suggested in [19] can be applied to specify OGDC; on how advanced features, such as coverage areas, can be modeled in Real-Time Maude; on additional analysis efforts; and on understanding the relationship between the results obtained by Real-Time Maude simulations and by ns-2 simulations. Lately, there has been some initial efforts applying Real-Time Maude to WSNs elsewhere [6, 20].

2 Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* [14] of the form (Σ, E, IR, TR) , where:

- (Σ, E) is a *membership equational logic* [10] theory with Σ a signature² and E a set of conditional equations. The theory (Σ, E) specifies the system’s state space as an algebraic data type. (Σ, E) must contain a specification of a sort `Time` modeling the time domain (which may be dense or discrete).
- IR is a set of *labeled conditional instantaneous rewrite rules* specifying the system’s *instantaneous* (i.e., zero-time) local transitions, each of which is written `cr1 [l] : t => t' if cond`, where l is a *label*. Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t' , *provided* the condition holds. The rules are applied *modulo* the equations E .³
- TR is a set of *tick (rewrite) rules*, written with syntax

`cr1 [l] : {t} => {t'} in time τ if cond .`

that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

² i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols* (or *operators*)

³ E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A . Operationally, a term is reduced to its E' -normal form modulo A before any rewrite rule is applied.

The initial states must be ground terms of sort `GlobalSystem` and must be reducible to terms of the form $\{t\}$ using the equations in the specifications. The form of the tick rules then ensures uniform time elapse in all parts of the system.

In object-oriented Real-Time Maude modules, a *class* declaration

```
class C | att1 : s1, ... , attn : sn .
```

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n . An *object* of class C in a given state is represented as a term $\langle O : C \mid att_1 : val_1, \dots, att_n : val_n \rangle$ where O is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n . In a concurrent object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has typically the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
r1 [1] : m(0,w) < 0 : C | a1 : x, a2 : 0', a3 : z > =>
          < 0 : C | a1 : x + w, a2 : 0', a3 : z > dly(m'(0'),x) .
```

defines a family of transitions in which a message m , with parameters 0 and w , is read and consumed by an object 0 of class C . The transitions have the effect of altering the attribute $a1$ of the object 0 and of sending a new message $m'(0')$ with delay x (see [17]). “Irrelevant” attributes (such as $a3$, and the *right-hand side occurrence* of $a2$) need not be mentioned in a rule.

Timed modules are *executable* under reasonable assumptions, and Real-Time Maude provides a spectrum of analysis capabilities. We summarize below the Real-Time Maude analysis commands used in our case study.

Real-Time Maude's *timed “fair” rewrite* command simulates *one* behavior of the system *up to a certain duration*. It is written with syntax

```
(tfrew t in time <= τ .)
```

where t is the initial state and τ is a ground term of sort `Time`.

Real-Time Maude's *timed search* command uses a breadth-first strategy to search for states that are reachable from a given initial state t within time τ , match a *search pattern*, and satisfy a *search condition*. The command which searches for *one* state satisfying the search criteria has syntax

```
(tsearch [1] t =>* pattern such that cond in time <= τ .)
```

Real-Time Maude also extends Maude's *linear temporal logic model checker* [3] to check whether each behavior “up to a certain time,” as explained in [17], satisfies a temporal logic formula. *State propositions* are terms of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form

```
{statePattern} |= prop = b
```

for b a term of sort `Bool`, which defines the state proposition $prop$ to hold in all states $\{t\}$ where $\{t\} \models prop$ evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, \sim (negation), \wedge , \vee , \rightarrow (implication), \square (“always”), \diamond (“eventually”), and U (“until”). The time-bounded model checking command has syntax

```
(mc t |=t formula in time <= τ .)
```

for t the initial state and *formula* the temporal logic formula.

Finally, the `find latest` command finds how long it takes, in the worst case, to reach a desired state.

3 Overview of the OGDC Algorithm

In a two-dimensional plane, a node with *sensing range* r_s can *sense* events in a circular *coverage area* with radius r_s . It is desirable that the coverage areas of the *active* nodes cover the entire area to be monitored (the “sensing area”) for as long as possible. A large number of nodes is often deployed to extend the lifetime of a wireless sensor network, so that some nodes can be intentionally “put to sleep” to save power. A node that is inactive can be switched on when needed. The process of periodically choosing the nodes that can be put to sleep while maintaining coverage (and connectivity) of the sensing area is called the *density control process*. The OGDC algorithm [22] is a state-of-the-art density control algorithm, developed by Zhang and Hou, that tries to select the set of active nodes such that their coverage areas provide the minimum amount of overlap.

The network lifetime is divided into *rounds*, where each round is divided into a *node selection phase* and a *steady state phase*. The node selection phase begins with each node having status “undecided” and *probabilistically* choosing whether or not to volunteer to be a *starting node*. Each node that volunteers sets its *backoff timer* to a small value. The node then becomes *active* when its backoff timer expires, and broadcasts a *power-on* message which contains the location of the node and a *random direction*. When an “undecided” node receives a power-on message, it checks if its entire coverage area is covered by the surrounding active nodes, in which case the node becomes inactive. Otherwise, it sets its backoff timer depending on how close the node is to the *optimal position* w.r.t. the nodes that are currently active. The timer value is set to a gradually larger value as the *distance increases* and the *direction deviates*. When the backoff timer of a node expires, the node becomes active and broadcasts a power-on message that may cause other nodes to reset their backoff timers or to become inactive. The network enters the steady state phase when each node is either active or inactive. When a round is over, the density control process starts over again.

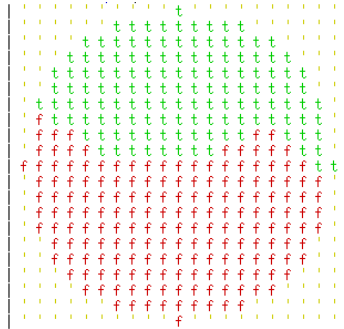


Fig. 1. The bitmap for a node’s coverage area.

4 The Real-Time Maude Specification of OGDC

This section presents a sample of our specification of the OGDC algorithm.⁴ General techniques for modeling typical WSN features, such as distances and communication, are described in [19].

4.1 Modeling Locations

We can represent a location in the plane as a term $x.y$, for rational numbers x and y , of the following sort `Location`:⁵

```
sort Location .
op _._ : Rat Rat -> Location .
```

4.2 Modeling Areas using Bitmaps

A significant part of the OGDC algorithm consists of checking whether a node’s coverage area is completely covered by the coverage areas of other active nodes, since this determines whether or not a node can be switched off. Zhang and Hou suggest in a preliminary version of [22] to use a bitmap to model a node’s coverage area. A coverage area is divided into a *grid*, and each bit in the bitmap represents the *center* of a grid square. The Real-Time Maude tool is not a graphical tool, but with proper use of the `format` operator attribute [3], a bitmap can be given an intuitive appearance as shown in Fig. 1. We define a bitmap as a term of sort `Bitmap`, which consists of a list of `BitLists`⁶, which in itself is a list of `Bits`. A `Bit` has one of three values: `t` if the location of the bit is covered by at least one other active node, `f` if the location is not covered, or the bit `'` that is used to “pad” the circles as shown in Fig. 1. The sort `Bitmap` is thus defined as follows:

⁴ Our specification is explained in detail in [21]. The entire executable Real-Time Maude specification can be found at <http://www.ifi.uio.no/RealTimeMaude/OGDC>.

⁵ Underbars in the declarations of operators such as `_._` denote the places of arguments for “mix-fix” function symbols.

⁶ Each `BitList` corresponds to a “row” in the bitmap.

```

sorts Bitmap BitList Bit .      subsort Bit < BitList .
ops t f ' : -> Bit [ctor] .
op nil : -> BitList [ctor] .
op -- : BitList BitList -> BitList [ctor assoc id: nil format (o s o)] .
op |_| : BitList -> Bitmap [ctor format (ni o o o)] .
op nil : -> Bitmap [ctor] .
op -- : Bitmap Bitmap -> Bitmap [ctor assoc id: nil] .

```

The location of each bit is computed from the location of the node which is the center of the bitmap. A function `updateBitmap` updates a node's bitmap when the node receives a *power-on* message (see rule `recPowerOn1`) by setting each bit within the sensing range of the sender to `t`. The node then also checks whether its (updated) bitmap is completely covered by its neighbors. This is done by the function `coverageAreaCovered`, which returns `false` if *some* bit is 'f' and returns `true` otherwise (*owise*):

```

vars BITL BITL' : BitList .   vars BM BM' : Bitmap .
op coverageAreaCovered : Bitmap -> Bool .
eq coverageAreaCovered(BM | BITL f BITL' | BM') = false .
eq coverageAreaCovered(BM) = true [owise] .

```

We choose to have 1 meter between each bit in a bitmap, which results in bitmaps with 400 bits (including the ' bits) since the sensing range of a node is 10 meters.

4.3 The Definition of Sensor Node Objects

We model sensor nodes as objects of the class `WSNode`. A sensor node does not have an explicit identifier but can be identified by its *location*. We let locations be object identifiers by giving the subsort declaration `subsort Location < Oid` .

```

class WSNode | backoffTimer : TimeInf, coverageArea : Bitmap,
               uncoveredCrossings : CrossingSet,
               remainingPower : Nat,   neighbors : NeighborSet,
               roundTimer : TimeInf,   status : Status,
               volunteerProb : Rat,    hasVolunteered : VolunteerStatus .

```

The attribute names are self-explanatory: `backoffTimer` denotes the time remaining until the node must perform an action; `coverageArea` contains the node's coverage area; `remainingPower` denotes the amount of power the node has left; `roundTimer` is the time remaining of the round; `status` denotes the node's status, which is either `on`, `off`, or `undecided`; `volunteerProb` gives the probability for the node to volunteer as a starting node; and `hasVolunteered` denotes whether the node has volunteered as a starting node.

4.4 Modeling Time and Time Elapse

We follow the guidelines in [17] for modeling time-dependent behaviors in object-oriented specifications. Time elapse is modeled by the tick rule

```

var C : Configuration .   var T : Time .
crl [tick] : {C} => {δ(C, T)} in time T if T <= mte(C) .

```

The function δ defines the effect of time elapse on a configuration, and the function `mte` defines the maximum amount of time that can elapse before some action must take place. These functions distribute over the objects and messages in a configuration and must be defined for single objects. The tick rule advances time nondeterministically by *any* amount T less than or equal to `mte(C)`. Before executing the system, a *time sampling strategy* guiding the application of the tick rule must be defined (see Section 5.1). We import the built-in module `NAT-TIME-DOMAIN-WITH-INF`, which defines the time domain `Time` to be the natural numbers, with an additional constant `INF` (for ∞) of a supersort `TimeInf`.

The function δ is defined on a `WSNode` object by decreasing its timers and amount of remaining power according to the time that has elapsed:

```

vars L L' : Location .   var T : Time .   vars TI TI' : TimeInf .
var P : NzNat .   var S : Status .   vars M N : Nat .   var D : Int .
var V : VolunteerStatus .   var R : Rat .   var NBS : NeighborSet .

eq δ(< L : WSNode | remainingPower : N, status : S,
      backoffTimer : TI, roundTimer : TI' >, T)
=
  < L : WSNode | remainingPower : if S == on then N monus (idlePower * T)
      else N monus (sleepPower * T) fi,
      backoffTimer : TI monus T, roundTimer : TI' monus T > .

```

The constants `idlePower` and `sleepPower` denote the amount of power the node consumes per time unit (millisecond) when the node is active and inactive, respectively. The function `monus` is defined by $x \text{ monus } y = \max(0, x - y)$.

The function `mte` is defined so that time cannot advance when a node is in its volunteering process (`undecided`)—forcing the node to enter this process at the start of each round—and otherwise cannot advance beyond the expiration time of a timer, or beyond the time when the node would run out of power:

```

eq mte(< L : WSNode | backoffTimer : TI, roundTimer : TI', status : S,
      remainingPower : P, hasVolunteered : V >) =
  if V == undecided then 0 else
    min(TI, TI', if S == on then ceiling(P / idlePower)
      else ceiling(P / sleepPower) fi) fi .

```

4.5 Modeling Communication

The informal description of the OGDC algorithm says that nodes *broadcast* messages within the radio range. Furthermore, a node does not know its neighbors. Most time related parameters in OGDC are set according to the *transmission time* of a message, which is assumed to be the same for all broadcast transmissions. This is a clear indication that transmission delays must be captured in the model. In [19] we show how such “area broadcast” with transmission delay Δ

can be easily modeled in Real-Time Maude. The idea is that the sender l sends a “broadcast message” `broadcast m from l` , where m is the message content, into the configuration. This broadcast message is then defined to be *equivalent* to a set of *single* messages `dly(msg m from l to l' , Δ)`, one such message for each sensor node l' within the radio range of l .

Since the description of OGDC does not discuss packet collisions, and only mentions that OGDC also should work in the presence of message losses, we have not modeled problems that are due to packet collisions.

4.6 Probabilistic Behaviors

The OGDC algorithm exhibits probabilistic behaviors in that (i) some actions are performed with probability p , and (ii) some values are supposed to be set to “random values, drawn from a uniform distribution ...” As mentioned, Real-Time Maude does not provide explicit support for specifying probabilistic behavior. Instead, for simulation purposes, we define a function `random`, which generates a sequence of numbers pseudo-randomly and which satisfies Knuth’s criteria for a “good” random number generator [7]. The state must then contain an object of a class `RandomNGen` with an attribute `seed` which stores the ever-changing “seed” for `random`. Probabilistic behaviors can then be modeled by “sampling” a value from the given interval using the `random` function. For the purpose of specifying all possible behaviors, we *could have*—but have not, due to the resulting large reachable state spaces that would have made exhaustive analysis unfeasible—modeled probabilistic behavior by nondeterministic behavior by (i) letting a probabilistic action be enabled as long as the probability of it being performed is greater than 0, and (ii) by letting the “random” value be a new variable, only occurring in the *right-hand* side of the rewrite rule, which can be given any value in the desired interval.

4.7 Defining the Dynamic Behavior of the OGDC Algorithm

The dynamic behavior of the OGDC algorithm is modeled in Real-Time Maude by 11 rewrite rules, 3 of which are given below.

At the start of each round of the OGDC algorithm, each node is in state `undecided` and must decide whether or not to volunteer as a *starting node*. This part of the protocol is described as follows in [22]:

A node volunteers to be a starting node with probability p if its power exceeds a pre-determined threshold P_t . [...] If a sensor node volunteers, it sets a backoff timer to τ_1 seconds, where τ_1 is uniformly distributed in $[0, T_d]$. When the timer expires, the node changes its state to “ON”, and broadcasts a power-on message. [...] The power-on message sent by the starting node contains (i) the position of the sender and (ii) the direction α along which the second working node should be located. This direction is randomly generated from a uniform distribution in $[0, 2\pi]$. [...] If the node does not volunteer itself to be a starting node, it sets a timer of T_s seconds. [...]

This part of the OGDC algorithm is probabilistic, since a node decides to volunteer with probability p . We simulate such probabilistic behavior in the following rewrite rules by checking whether the next pseudo-random number generated in the system, modified to a value between 0 and 999 (`randomProb(M)`, defined as `random(M) rem 1000`), is less than R , where R denotes the current volunteering probability multiplied by 1000. The first rule models the start of the “starting node selection” phase when the node’s `hasVolunteered` attribute is `undecided`:

```

r1 [volunteer] :
  < L : WSNODE | remainingPower : P, volunteerProb : R,
      hasVolunteered : undecided >
  < Random : RandomNGen | seed : M >
=>
  (if (randomProb(M) < R) and (P > powerThreshold or R == 1000)
    then < L : WSNODE | backoffTimer : randomTimer(random(M)),
        hasVolunteered : true >
    else < L : WSNODE | backoffTimer : Ts, hasVolunteered : false,
        volunteerProb : doubleProb(R) >
    fi)
  < Random : RandomNGen | seed : random(random(M)) > .

```

The node must also have sufficient remaining power ($P > \text{powerThreshold}$), or its volunteer probability must have reached 1 ($R == 1000$). If the node volunteers, it sets its backoff timer to a random value between 0 and T_d by the function `randomTimer`. If the node does not volunteer, it sets its backoff timer to T_s . The `seed` is also updated, so that the next application of this (or any other) rule will draw a completely different random number.

A node becomes *active* when its backoff timer expires. If the node volunteered as a starting node, it broadcasts a power-on message that contains the node’s location and a *random direction*:

```

r1 [startingNodePowerOn] :
  < L : WSNODE | remainingPower : P, backoffTimer : 0,
      hasVolunteered : true >
  < Random : RandomNGen | seed : M >
=>
  < L : WSNODE | remainingPower : P minus transPower,
      backoffTimer : INF, status : on >
  < Random : RandomNGen | seed : random(M) >
  broadcast (powerOnWithDirection randomDirection(M)) from L .

```

The node consumes `transPower` amount of power when it broadcasts a message.

The actions taken when a node receives a power-on message are described as follows in [22]:

When a sensor node receives a power-on message, if the node is already “ON”, or it is more than $2r_s$ away from the sender node, it ignores the message; otherwise it adds this node to its neighbor list, and checks whether or not all its neighbors’ coverage disks completely cover its own coverage disk. If so, the node sets its state to “OFF” and turns itself off. Otherwise [...]

The next rule models the case where the receiver has status `undecided` and its coverage area becomes entirely covered by its active neighbors (including the sender of the current power-on message). In this case, the node turns itself `off`:

```

crl [recPowerOn1] :
  (msg (powerOnWithDirection D) from L' to L)
  < L : WSNode | status : undecided, neighbors : NBS, bitmap : BM >
=>
  < L : WSNode | status : off, neighbors : NBS (L' starting (D >= 0)),
    bitmap : updateBitmap(L, BM, L'), backoffTimer : INF >
  if (L withinTwiceTheSensingRangeOf L')
    /\ coverageAreaCovered(updateBitmap(L, BM, L')) .

```

5 Simulation and Formal Analysis of OGDC

This section describes how the OGDC algorithm can be subjected to the following kinds of formal analysis in Real-Time Maude:

1. *Monte Carlo simulation*, with probabilistic behavior simulated using our pseudo-random number generator, by *timed fair rewriting*. In particular, we show how Real-Time Maude can perform *all* the simulations done by Zhang and Hou on the wireless extension of the network simulation tool ns-2.
2. Time-bounded *reachability analysis* and *temporal logic model checking* of all possible behaviors from some initial state *with respect to the particular values generated by the pseudo-random generator*. That is, our analysis is *incomplete* since we do *not* analyze all possible behaviors for a given network topology, but only those that can take place with the specific choice of pseudo-random numbers used to simulate the probabilistic behavior. Nevertheless, such analysis covers *many* different behaviors from a given state.

In our experiments, we use the same values for parameters such as sensing range (10m), length of a round (1000 seconds), power consumption, transmission times, etc., as in the ns-2 simulations in [22]. In those simulations, 100 to 1000 nodes were “uniformly randomly distributed” in a $50m \times 50m$ sensing area.

5.1 Defining Initial States and the Time Sampling Strategy

To easily simulate large sensor networks with different node locations and initial seeds, we define a function `genInitConf` to generate initial states. The term `genInitConf(n, seed)` defines a configuration with n sensor nodes scattered at pseudo-random locations within the sensing area, as well as a `RandomNGen` object with starting seed computed from the initial seed `seed`. (An initial state must also add the operator `{_}`.) We can therefore generate initial states with any number of nodes, and/or place them in different locations, by just changing the parameters n and/or `seed` in `genInitConf`.

In the following definition, each generated sensor node location $x.y$ will have $0 \leq x \leq Xsize$ and $0 \leq y \leq Ysize$ (since `rem` is the remainder function):

```

op genInitConf : Nat Nat -> Configuration .
op genInitConf : Nat Nat Nat -> Configuration .

vars M SEED N : Nat .

eq genInitConf(N, SEED) = genInitConf(N, SEED, N) .

ceq genInitConf(M, SEED, N) =
  if M == 0 then
    --- no more nodes to generate; generate RandomNGen object:
    < Random : RandomNGen | seed : SEED >
  else --- more nodes to generate:
    < L : WSNODE | remainingPower : lifetime, status : undecided,
                  neighbors : none, bitmap : initBitmap(L),
                  uncoveredCrossings : none, backoffTimer : INF,
                  roundTimer : roundTime, volunteerProb : 1000 / N,
                  hasVolunteered : undecided >
    --- and generate the remaining M-1 nodes:
    genInitConf(M - 1, random(random(SEED)), N)
  fi
  if L := random(SEED) rem (Xsize + 1) . --- x part of L
    random(random(SEED)) rem (Ysize + 1) . --- y part of L

```

Each generated `WSNode` gets the appropriate initial values for its attributes. The third argument to the `genInitConf` in the main equation is needed to store the total number of nodes in the system (`N`) so that the `volunteerProb` attribute gets the correct initial value.

A time sampling strategy guiding the execution of the tick rule must be chosen before any analysis can take place. Since all events in the OGDC algorithm happen at specific times, we have shown in [16] that we can “fast forward” between these events without losing any interesting behaviors. Therefore, in our analysis, we use the *maximal* time sampling strategy declared by the Real-Time Maude command (`set tick max def roundTime .`) which advances time as much as possible, and corresponds to “event-driven simulation.”

5.2 The ns-2 Simulations of OGDC in Real-Time Maude

In [22], Zhang and Hou use the network simulation tool *ns-2* [12], with the wireless extension developed by the CMU Monarch group [4], to simulate the OGDC algorithm and measure the following essential *performance metrics*:

- The number of *active* nodes and the percentage of sensing area coverage provided by those nodes at the end of the first round.
- The percentage of sensing area coverage and the total amount of remaining power for the whole system throughout the network’s lifetime.
- The total time during which at least α percent of the sensing area is covered. (This can be done in the same way as the first two, and is not treated here.)

We cannot use Real-Time Maude's timed rewrite command *directly* to perform the corresponding analysis, since these performance metrics should be measured at different points in time *throughout* the lifetime of the system, and since the metrics themselves do not appear explicitly in the state. Therefore, we add to the initial state a *record object* that uses a timer to compute a performance metric at the same time (e.g., just before the end of the round) in each round during a simulation of the OGDC algorithm. The computed values are stored in an attribute of the record object as a list $n_1 ++ n_2 ++ \dots ++ n_k$, where n_i denotes the value of the metric at the end of round i . Given a sort `NatList` of lists of natural numbers, with concatenation operator `_++_` and empty list `nil`, we can declare the record object class as follows:

```
class RecActNodes | activeNodes : NatList, timer : TimeInf .
ops r1 r2 r3 : -> Oid [ctor] . --- names of record objects
```

The following rule applies when the `timer` of the record object expires. It computes and stores the number of active nodes in the system, and *resets* the `timer` in order for it to be expire again at the same time in the *next* round:

```
var SYSTEM : Configuration . var NL : NatList . var O : Oid .
r1 [computeNumActiveNodes] :
  {< O : RecActNodes | activeNodes : NL, timer : O > SYSTEM}
=>
  {< O : RecActNodes | activeNodes : NL ++ numActiveNodes(SYSTEM),
    timer : roundTime > SYSTEM} .
```

The function `numActiveNodes` computes the number of active nodes in a configuration. In the same way, we define record object classes `RecCoverage%` and `RecTotalPower`, which compute, respectively, the percentage of the sensing area covered by the active nodes and the total amount of power in the system.

The first simulations in [22] investigate the number of active nodes and the percentage of coverage in the *first* round of the algorithm. The following timed fair rewrite command simulates a system with 600 nodes (in a $50m \times 50m$ sensing area) until the end of the first round of the protocol (`in time < roundTime`). The initial state contains two record objects, whose metrics will be computed when their `timers` expire just before the end of the first round (`roundTime - 1`):

```
Maude> (tfrew {genInitConf(600, 1)
  < r1 : RecActNodes | activeNodes : nil,
    timer : roundTime - 1 >
  < r2 : RecCoverage% | cov% : nil, timer : roundTime - 1 >}
in time < roundTime .)
```

```
Result ClockedSystem :
  {< r1 : RecActNodes | activeNodes : 45, timer : 1000000 >
  < r2 : RecCoverage% | cov% : 100, timer : 1000000 >
  ... } in time 999999
```

As shown in the analysis messages, 45 of the 600 deployed nodes became active nodes and together provided 100% coverage of the sensing area.

Zhang and Hou then measure how coverage and total remaining power changes over time. The following rewrite command simulates 50 rounds of the algorithm (`in time < roundTime * 50`) with 200 nodes in the $50m \times 50m$ sensing area:

```
Maude> (tfrew {genInitConf(200, 313)
          < r1 : RecCoverage% | cov% : nil, timer : roundTime - 1 >
          < r2 : RecTotalPower | power : nil, timer : roundTime - 1 >}
      in time < roundTime * 50 .)
```

```
Result ClockedSystem :
{< r1 : RecCoverage% | cov% : 100 ++ ... ++ 100 ++ 98 ++ ... ++ 100 ++ 94
    ++ 88 ++ ... ++ 13 ++ 0 ++ ... ++ 0), ... >
 < r2 : RecTotalPower | power : 384639803547 ++ 370475585958 ++ ...
    ++ 371677818 ++ 0 ++ ... ++ 0), ... >
... } in time 49999999
```

The result shows that the nodes can provide 100% coverage for 19 rounds, with a decrease of coverage in certain intermediate rounds.

5.3 Comparison with the ns-2 Simulations

The table on the next page compares our simulation results with the ns-2 simulation results in [22]. Our simulations show a higher number of active nodes (more than twice as many, in fact) and a correspondingly shorter network lifetime. Furthermore, in contrast to the ns-2 simulations, we get more active nodes when more nodes are deployed in the same area. These differences cannot be explained by us ignoring packet collisions in our simulations, since [22] states that “the number of working nodes may *increase*” in the presence of message losses. The most plausible explanation for the different results is instead the following: In OGDC, if two nodes are close to one another, then the difference between their backoff timers is smaller than the transmission delay. If transmission delays are ignored during the simulations, potentially because the simulation tool makes it inconvenient to simulate such delays, then only one of the neighbors will become active. However, if, as in our case, we capture transmission delays, then the backoff timer of the “worse” node will expire *before* it receives the power-on message from the “better” node, and, hence, *both* nodes will become active.

We have, unfortunately, not been able to get an answer to whether or not the ns-2 simulations in [22] actually took the transmission delays into account, although the second author told us it is quite likely that they did not. Therefore, we have also performed the simulations *without* transmission delays (by just removing the `dly`-part from the single messages created by the broadcast). The following table shows the results of the ns-2 simulations, as well as of the Real-Time Maude simulations both with and without transmission delays, for finding the number of active nodes at the end of the first round for 200, 400, and 600 nodes in the same $50m \times 50m$ area. For the Real-Time Maude simulations, each

number represents the average result of five simulations, obtained by using five different initial seeds (and hence getting five different placements of the nodes):

Number of nodes in sensing area	200	400	600
# active nodes in ns-2 simulations	17	18	18
# active nodes in Real-Time Maude simulations <i>with</i> trans. delay	34	45	55
# active nodes in Real-Time Maude simulations <i>without</i> trans. delay	21	22	22

Indeed, the results of the Real-Time Maude simulations that ignore transmission delays are quite similar to the results of the ns-2 simulations. It is therefore tempting to conjecture that our Real-Time Maude simulations *with* transmission delays give a reasonably accurate estimate of the performance of OGDC in such a setting. In that case, one can conclude that the results of ns-2 simulations are actually quite misleading and that our formal model provides a more accurate simulation setting for OGDC than ns-2 with the wireless extension.

5.4 Further Real-Time Maude Analysis of the OGDC Algorithm

We give some examples of how we can further formally analyze correctness and worst-case performance of the OGDC algorithm by using Real-Time Maude’s search and model checking capabilities. Due to the large states involved, we restrict such analyses to systems with 5 to 6 nodes (in a $25m \times 25m$ area), which is much fewer nodes than in a real WSN. Nevertheless, exhaustive analysis with 3 to 4 nodes has uncovered subtle bugs in cryptographic protocols [9] and other kinds of network protocols (e.g., [18]).

The following `find latest` command finds the *latest* possible time the network enters the steady state phase (`such that steadyStatePhase(...)`), and thereby also finds out whether this phase is *always* reached in the first round.

```
Maude> (find latest {genInitConf(6, 75)} =>* {C:Configuration}
      such that steadyStatePhase(C:Configuration)
      in time < roundTime .)
```

```
Result: { ... } in time 372
```

That is, the system will reach the steady state phase in *at most* 372 ms. One round of the OGDC algorithm is 1000 *seconds*, which means that the network spends most of its lifetime performing its sensing task.

Another correctness requirement is that the network stays in the steady state phase throughout the first round, once this phase has been reached. We use Real-Time Maude’s temporal logic model checker, and define an atomic proposition `steady-state` to hold when the network is in steady state phase:

```
op steady-state : -> Prop [ctor] .
eq {C} |= steady-state = steadyStatePhase(C) .
```

The following command checks whether all states following a state in the steady state phase are also in this phase ($A \Rightarrow B$ is an abbreviation for $[] (A \rightarrow B)$).

```
Maude> (mc {genInitConf(5,341)} |=t (steady-state => [] steady-state)
      in time < roundTime .)
```

Result Bool : true

The most important correctness criterion is that the entire sensing area is covered by the *active* nodes when the system is in the steady state phase (and *all* nodes together cover the entire area and each node has power to last to the end of the round). The following command searches for a state which is in steady state but where the entire $20m \times 20m$ sensingArea is *not* covered by the active nodes:

```
Maude> (tsearch [1]
      {genInitConf(5,97)} =>* {C:Configuration}
      such that steadyStatePhase(C:Configuration) /\
      not coverageAreaCovered(updateArea(sensingArea,
      C:Configuration))
      in time < roundTime .)
```

The function `updateArea` updates the bitmap by changing bits that are covered by the *active* nodes in `C` to `t`. The command returned ‘No solution.’

Performance figures. The following table shows, for each command presented in this paper, and for the above search command with a different topology (given by seed 1) which does *not* cover the sensing area: the number of sensor nodes; execution time; and memory usage when executed on a 3.6 GHz Intel Xeon:

tfrew 1 rd	tfrew 1 rd	tfrew 1 rd	tfrew 50 rds	find	mc	tsearch	tsearch
200	400	600	200	6	5	6, s=1	5, s=7197
180 sec	1243 sec	5034 sec	4931 sec	4187 sec	26 sec	679 sec	227 sec
85 MB	100 MB	112 MB	93 MB	525 MB	147 MB	1.3 GB	430 MB

The paper [22] does not mention the performance of their ns-2 simulations.

6 Concluding Remarks

Wireless sensor networks are a new kind of network whose modeling, simulation, and/or analysis pose a set of challenges to both network simulation tools and formal tools. OGDC is a state-of-the-art WSN algorithm where new forms of communication and advanced data types must be captured at an appropriate level of abstraction. In this paper we have shown how OGDC was formally specified, simulated, and analyzed using Real-Time Maude. To the best of our knowledge, this is the first formal analysis of an advanced WSN algorithm. Our formal specification captures the behavior of the algorithm at a high level of abstraction and—being precise, intuitive, and operational—could make a good starting point for an implementation of the OGDC algorithm on sensor networks.

We could measure *all* performance metrics measured in the ns-2 simulations in [22] during our “Monte Carlo” simulations. Our simulations showed significantly worse performance of the OGDC algorithm than the ns-2 simulations.

Trying to understand why—unlike in the ns-2 simulations—we got more active nodes when more nodes were deployed in the same sensing area, we found that the “tie-breaking” mechanism in OGDC does not break many ties when transmission delays are taken into account. To check this hypothesis, we also simulated OGDC in Real-Time Maude in a setting *without* transmission delays, and got results that were similar to the ns-2 results. It is therefore quite likely that our simulations, which take the delays into account, provide much more accurate performance estimates than the ns-2 simulations that may have ignored such delays. Furthermore, based on communication with Jennifer Hou, it seems that developing the Real-Time Maude specification and performing the Real-Time Maude analysis required much less effort than using a specialized network simulation tool to analyze OGDC.

Our work should continue in different directions. First, we focus on simplicity and elegance when modeling coverage areas and defining functions on such areas. There is a price to pay for this when we have hundreds of nodes, each with a bitmap with 400 “bits.” Therefore, more efficient representations of coverage areas should be developed. This would enable us to perform search and model checking on larger networks.

Second, we have not modeled probabilistic behaviors as such, but have used a “sampling” technique for simulation purposes. This means that we cannot reason about probabilistic properties. We should therefore combine Real-Time Maude with methods and tools for probabilistic systems, such as PMaude [1], and should develop methods to fruitfully analyze probabilistic real-time specifications.

Finally, we should also capture message losses due to packet collisions.

Acknowledgments. We are grateful to Jennifer Hou for suggesting the OGDC algorithm as a challenging modeling task, and for discussions on sensor networks, to José Meseguer for discussions on modeling communication in sensor networks, and to the anonymous reviewers for helpful comments on earlier versions of this paper. Support by the Research Council of Norway is also gratefully acknowledged.

References

1. G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *Proc. QAPL'05*, 2005.
2. I.F. Akyildiz, W. Su, Y. Sankarasubramanian, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38:393–422, 2002.
3. M. Clavel, F. Dúran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, December 2005. <http://maude.cs.uiuc.edu>.
4. CMU monarch extensions to ns. <http://www.monarch.cs.cmu.edu/>.
5. S. Coleri, M. Ergen, and T. J. Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *WSNA '02*. ACM, 2002.
6. M. Kim, N. Dutt, and N. Venkatasubramanian. Policy construction and validation for energy minimization in cross layered systems: A formal method approach. In *IEEE RTAS'06 Work-in-Progress Session*, pages 25–28, 2006.
7. D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, second edition, 1981.

8. E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Dept. of Linguistics, University of Oslo, 2004.
9. G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.
10. J. Meseguer. Membership algebra as a logical framework for equational specification. In *WADT'97*, volume 1376 of *LNCS*. Springer, 1998.
11. S. Nair and R. Cardell-Oliver. Formal specification and analysis of performance variation in sensor network diffusion protocols. In *MSWiM '04*. ACM, 2004.
12. ns-2 network simulator. <http://www.isi.edu/nsnam/ns>.
13. P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In L. Baresi and R. Heckel, editors, *FASE'06*, volume 3922 of *LNCS*, pages 357–372. Springer, 2006.
14. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.
15. P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In *FASE 2004*, volume 2984 of *LNCS*. Springer, 2004.
16. P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. In *Proc. WRLA'06*, 2006.
17. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 2007. To appear.
18. P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29:253–293, 2006.
19. P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *IPDPS 2006*. IEEE, 2006.
20. D. E. Rodríguez. On modelling sensor networks in Maude. In *Proc. WRLA 2006*.
21. S. Thorvaldsen and P. C. Ölveczky. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. Manuscript. <http://www.ifi.uio.no/RealTimeMaude/OGDC>, October 2005.
22. H. Zhang and J. C. Hou. Maintaining sensing coverage and connectivity in large sensor networks. *Wireless Ad Hoc and Sensor Networks: An International Journal*, 1, 2005.

A Modeling Communication and Defining Initial States

One of the key reasons enabling us to model OGDC at (what we consider to be) an appropriate level of abstraction is that different forms of communication, including the one assumed in OGDC, can be easily defined in Real-Time Maude. In [19] we show how to model typical WSN broadcast communication. Since communication plays a crucial role in formally modeling OGDC, we explain this again in Section A.1.

A.1 Modeling Communication in WSNs

The following function `distanceSq` defines the *square* of the distance between two locations:⁷

⁷ Real-Time Maude also provides a built-in data type of floating-point numbers, with functions such as square root, but we prefer to stay within the rational numbers whenever possible.

```

op distanceSq : Location Location -> Rat .
vars X X' Y Y' : Rat .
eq distanceSq(X . Y, X' . Y) =
  ((X - X') * (X - X')) + ((Y - Y') * (Y - Y')) .

```

Given a constant `transRange` denoting the transmission range of a sensor node, we can check whether (the location of) a node is within the transmission range of another node:

```

vars L L' : Location .
op _withinTransRangeOf_ : Location Location -> Bool .
eq L withinTransRangeOf L' =
  distanceSq(L, L') <= transRange * transRange .

```

Assuming that object identifiers are locations, we can now define a communication model for WSNs as it seems to be assumed in the informal description of OGDC given in [22]. That description says that nodes *broadcast* messages within the radio range. Furthermore, a node does not know its neighbors. Most time related parameters in OGDC are set according to the transmission time of a message, which is a clear indication that transmission delays must be captured in the model. In OGDC, the transmission delay does not depend on the distance between sender and receiver. We have *not* modeled packet collisions.

In what follows, we model broadcast where a message must reach all nodes within the radio range of the sender and where the transmission is subject to a transmission delay Δ . The idea is that the sender l sends a “broadcast message” *broadcast m from l* , where m is the message content, into the configuration. This broadcast message is defined to be *equivalent* to a set of single messages `dly(msg m from l to l' , Δ)` with delay Δ , one for each sensor node l' within the radio range of l . The messages are declared as follows:

```

sort MsgCont . --- Message content
msg broadcast_from_ : MsgCont Location -> Msg .
msg msg_from_to_ : MsgCont Location Location -> Msg .

```

The following equation defines the desired equivalence:

```

var C : Configuration . var MC : MsgCont .
eq {< L : WSNode | > (broadcast MC from L) C} =
  {< L : WSNode | > distributeMsg(L, MC, C)} .

```

It is the task of `distributeMsg` to create an addressed message for each `WSNode` object in `C` that is within the transmission range of `L`. The use of the operator `{_}` enables the equation to grab the *entire* state (`C`), except the sender `L`, to ensure that *all* appropriate nodes in the system will get the message. The function `distributeMsg` is defined recursively over the elements in a configuration:

```

op distributeMsg : Location MsgCont Configuration
  -> Configuration [frozen (3)] .

```

```

var MSG : Msg .   var O : Oid .   var OBJECT : Object .
eq distributeMsg(L, MC, none) = none .

eq distributeMsg(L, MC, MSG C) =
  MSG distributeMsg(L, MC, C) .

eq distributeMsg(L, MC, < L' : WSNODE | > C) =
  < L' : WSNODE | > distributeMsg(L, MC, C)
  if L withinTransRangeOf L'
  then dly(msg MC from L to L', Δ) else none fi .

eq distributeMsg(L, MC, OBJECT C) =
  OBJECT distributeMsg(L, MC, C) [owise] .

```

The first equation above distributes the message from L with content MC to the empty configuration `none`. The second equation distributes the message to another message MSG and the remaining part C of the configuration. No new message should be created in these cases. The third equation distributes the message to a configuration consisting of a `WSNode` object L' and the remaining configuration C . In this case, a single message to L' is created if L' is within the transmission range of L . Finally, the fourth equation distributes the message to objects that are not `WSNode` objects (attribute `owise`), that is, to the `RandomNGen` object or to a record object used in the simulations. A new message to such an object is not created. In the last three equations, the message content is then also recursively distributed to the remaining part C of the configuration.

To illustrate the flexibility of the language, we see that if the transmission delay between two nodes l and l' is instead a function of the distance between them, say $f(l, l')$, we can just replace Δ with $f(L, L')$ in the last equation. This flexibility allowed us, as mentioned, to simulate OGDC in a setting *without* transmission delays by just setting Δ to be 0 (or, equivalently, by removing the `dly` part of the third equation above).

A.2 More Details About Our Simulation Results

In Section 5.3 we present the average of five Real-Time Maude simulations for different number of nodes in the system. In what follows, we spell out the number of active nodes found at the end of the first round for each of the separate Real-Time Maude simulations, both with and without transmission delays.

Real-Time Maude simulation with transmission delays. The following table shows the number of active nodes in our simulations *with transmission delays*, for different values of n and *seed* in the initial state (`genInitConf`):

Number of nodes in sensing area	200	400	600
# active nodes for seed=1	32	38	45
# active nodes for seed=5	38	47	51
# active nodes for seed=97	26	46	58
# active nodes for seed=313	38	54	60
# active nodes for seed=341	37	39	61

Real-Time Maude simulation without transmission delays. The following table shows the number of active nodes at the end of the first round in our simulations *without transmission delays*:

Number of nodes in sensing area	200	400	600
# active nodes for seed=1	21	25	22
# active nodes for seed=5	19	20	24
# active nodes for seed=97	19	23	25
# active nodes for seed=313	21	23	21
# active nodes for seed=341	23	20	18

ns-2 simulation results reported in [22]. The average of 20 ns-2 simulations for different number of nodes are presented in a diagram in [22]. It is somewhat difficult to see the exact values in this graphic representation, but to me it looks like the number of active nodes they get is around 17-19, and is independent of the number of sensor nodes deployed.

This is fairly different from the results of our Real-Time Maude simulations *with* transmission delays, which, in addition to reporting significantly more active nodes, also report more active nodes when more nodes are deployed. On the other hand, this is not the case (to a significant degree) in our Real-Time Maude simulations *without* transmission delays, which report results that can be considered fairly close to the ns-2 simulation results.