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

Stian Thorvaldsen and Peter Csaba Ölveczky

Department of Informatics, University of Oslo

October 20, 2005

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. The OGDC algorithm poses many challenges to its formal specification and analysis, including novel communication forms, treatment of geographical areas, time-dependent and probabilistic features, and the need to analyze both correctness and performance. This paper explains how we formally specified OGDC, using sampling techniques to simulate probabilistic behaviors. We show how we could simulate our specification to perform all the different analyses done by the algorithm developers using the network simulation tool ns-2. We also show how our specification can be subjected to state space exploration analysis.

1 Introduction

This paper describes the application of Real-Time Maude [29, 27, 30] to the formal specification, simulation, and further formal analysis of the sophisticated state-of-the-art OGDC wireless sensor network algorithm [37]. To the best of our knowledge, our work represents the first formal modeling and analysis effort of such a complex wireless sensor network system.

A wireless sensor network consists of a set of small, cheap, and low-power sensor nodes that use wireless technology to communicate with each other. The sensor nodes are equipped with sensing technology that enables them to observe different aspects of their surrounding environment, such as motion, heat, sound, etc. Wireless sensor networks have opened up a new range of applicable areas where environment observation and computer networking is desirable. Such areas can be, e.g., battlefield surveillance for targeting of enemy forces, measuring the condition of soil, monitoring forests for fires, or, when the sensor nodes become sufficiently small, measuring a person's medical conditions by placing the sensor nodes in the person's blood stream [2]. Sensor nodes tend to have limited power supply (usually provided by a battery) that is often virtually impossible to replace, since the nodes may be deployed in areas that are hard to reach,[1] such as on a battlefield or in the middle of a forest fire. Consequently, a wireless sensor network has limited lifetime.

With the emergence of a new kind of computer system such as wireless sensor networks, there is a need to be able to formally specify and analyze such systems. In this work we investigate the suitability of using Real-Time Maude for the formal modeling and analysis of wireless sensor networks. Real-Time Maude is a high-performance tool that extends the rewriting logic-based Maude system [9, 10] to support the formal specification and analysis of object-based real-time systems. Real-Time Maude emphasizes ease and expressiveness of specification, and provides a spectrum of analysis methods, including symbolic simulation through timed rewriting, time-bounded temporal logic model checking, and time-bounded and unbounded search for reachability analysis. Real-Time Maude complements formal real-time tools such as the timed/hybrid automaton-based tools Uppaal [4], Kronos [35], and Hytech [16] by having a more expressive specification formalism which supports well the specification of "infinite-control" systems which cannot be specified by such automata. On the other hand, Real-Time Maude can also be seen

[1] Indeed, it is often the purpose of sensor networks to monitor areas which cannot be easily monitored in other ways.

as complementing simulation tools, such as, e.g., ns-2 [23], GloMoSim [36], and JiST [3] by providing a precise specification at a high level of abstraction which can be simulated and further analyzed in different ways. Real-Time Maude has proved useful for analyzing sophisticated communication protocols [24, 25, 12, 19] and scheduling algorithms [31]. Real-Time Maude analysis has discovered subtle design errors in such protocols that were not discovered during traditional testing [25, 12].

Jennifer Hou recently suggested to us the *optimal geographical density control* (OGDC) algorithm for wireless sensor networks as a challenging modeling and analysis task for Real-Time Maude. The OGDC algorithm tries to maintain complete sensing coverage and connectivity for as long time as possible by switching nodes on and off. It has been simulated in the simulation tool *The Network Simulator* (ns-2) [23], where its performance was compared to the performance of similar algorithms.

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

1. Modeling—and computing with—geometric enties such as coverage areas, angles, and distances.
2. Modeling broadcast communication with limited range in a setting where a node does not know its neighbors, and where broadcast is subject to transmission delays.
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 scattered randomly in the sensing area.
6. Both correctness and, in particular, performance are critical aspects that must be analyzed.

This is a tall order. Real-Time Maude allowed us meet challenge (1) by defining the appropriate data types as equational specifications. Our specification of areas emphasizes ease and elegance of specification over computational efficiency. Real-Time Maude supports an object-based specification style which is ideal for modeling a network system with real-time features, and we could easily define the appropriate communication form as an extension of Maude's traditional message passing model [21]. We have *not* modeled the probabilistic behaviors as such, since that would require a different extension of rewriting logic [1], but have used sampling techniques with pseudo-random number generators for the purpose of *simulating* probabilistic behaviors. Regarding challenges (5) and (6), we could very easily define initial states with any number of nodes placed at pseudo-random locations, and could simulate systems with hundreds of sensor nodes. We introduced *analysis messages* to take snapshots of critical performance metrics during a simulation of the algorithm, and were able to do in Real-Time Maude *all* the analysis that Zhang and Hou performed using the wireless extension of the network simulation ns-2 [23]. We have also subjected the system to *time-bounded* explicit-state reachability analysis and temporal logic model checking. Such analyses normally explore *all* the behaviors of the system up to a certain time bound, but in our case they are also relative to the sampling techniques used for simulating probabilistic behaviors.

Related work: *Still to do . . . , but not aware of much except of hybrid automata modeling.*

The rest of the paper is organized as follows. Section 2 gives a brief overview of the OGDC algorithm. Section 3 introduces Real-Time Maude's specification language and analysis capabilities. Section 4 presents the Real-Time Maude model of the OGDC algorithm. In section 5 we show how Real-Time Maude can perform simulations corresponding the ns-2 simulations described in [37], and how, in addition, our tool can perform further analysis. The paper is concluded in Section 6.

2 The Optimal Geographical Density Control Algorithm

It is important in a wireless sensor network that the sensor nodes do not waste their power, but collaborate to maintain the network *operational* for as long time as possible. By operational we mean that the network provides *sensing coverage* and *connectivity* of the entire area to be monitored ("the *sensing area*"). A large number of nodes is often deployed to extend the operational lifetime of a wireless sensor network. In

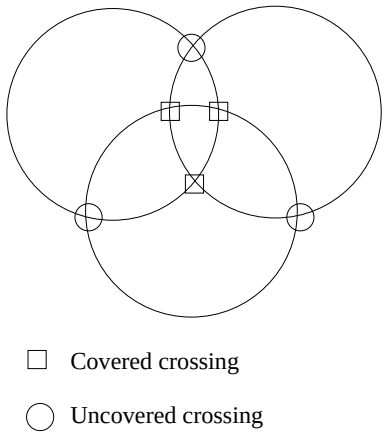


Figure 1: Covered and uncovered crossing.

that way, not all the nodes need to be *active* all the time, and some nodes can be intentionally switched off to save power. A node that is switched off can be switched on when needed. The process of choosing the nodes that can be switched off while maintaining coverage and connectivity of the sensing area is called the *density control process*.

The *optimal geographical density control* algorithm (OGDC) [37] is a sophisticated state-of-the-art density control algorithm developed by Zhang and Hou. It periodically selects nodes to be active and inactive in order to maintain sensing coverage of the entire sensing area while keeping a minimum number of active nodes. The OGDC algorithm is a fully *localized* distributed algorithm in the sense that each node uses only local information to carry out the density control process.

The OGDC algorithm assumes that sensor nodes are equipped with small radio transmitters and communicate by *broadcasting*. The broadcast works with *limited* signal strength, which implies that only nodes that are within a given distance from the sender will receive the broadcast with sufficient signal strength.

In [37], the authors make the following reasonable assumptions to focus on the central parts of the algorithm:

- Position awareness.
- The nodes are time synchronized.
- The radio range is at least twice as large as the sensing range.[2]

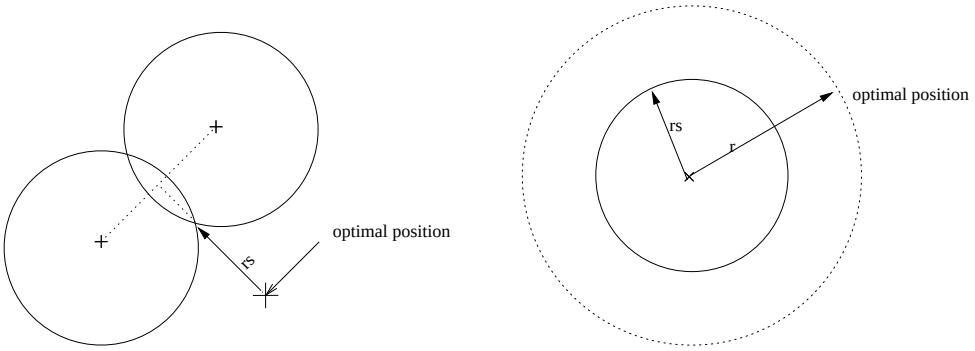
The two first assumptions fall outside of the scope of the OGDC algorithm. We may assume that some localization [8, 33, 7] and time synchronization [15, 17, 14] protocols have already been used prior to OGDC. It is proved in [37] that coverage implies connectivity when the radio transmission range is at least twice the sensing range. According to the same paper, having the radio range *less* than twice the sensing range is more the exception than the rule. This makes our third assumption reasonable, and allows us to focus on coverage only.

The intersection of the boundaries of the coverage areas[3] of two *active* nodes is called a *crossing*. This crossing is *uncovered* if it is not within the coverage area a third active node (see Fig. 1). According to [37], coverage is guaranteed if there exists at least one crossing in the sensing area and all crossings are covered. Furthermore, minimizing the number of active nodes is the same as minimizing the *overlap* of the coverage areas of all the active nodes.

The OGDC algorithm tries to select the set of active nodes such that they provide the minimum amount of overlap while leaving no crossing uncovered. The term *optimal position* denotes the location where a node *ideally* should be placed, with respect to the active nodes that are selected so far, to achieve this (see Fig. 2). Since the nodes are not carefully placed, a node is not always found at a given optimal position. Therefore, the OGDC algorithm selects the node that is *closest* to a given optimal position

[2] Zhang and Hou discuss how their algorithm can be extended when this assumption does not hold.

[3] The coverage area of a node denotes the disk-shaped area which is within the sensing range r_s of the node.



(a) The optimal position with respect to an uncovered crossing. *rs* is the radius of a node's coverage area.

(b) The optimal position with respect to a single starting node is a random point on a circle with radius $r = \sqrt{3}rs$

Figure 2: Optimal position

by the use of *backoff timers*. In essence, when a node receives a packet, it computes how far away it is from a perceived optimal position. The closer a node is to an optimal position, the smaller its backoff timer value becomes. In this way, a node in good position will become active before the backoff timer of a node in a worse location expires, and the packet broadcast by the well-positioned node upon its activation may inhibit the other node from becoming active.

2.1 Overview of the OGDC Algorithm

The network lifetime is divided into *rounds*, where each round has two phases:

- the *node selection* phase, and
- the *steady state* phase.

The node selection phase starts at the beginning of each round of the OGDC algorithm, and is the phase where the set of active nodes is selected. The steady state phase is the phase where the set of active nodes has been chosen, and the network can perform its sensing task.

The node selection phase begins with each node entering a *volunteering process* where it probabilistically chooses 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 as a starting node when its backoff timer expires, and broadcasts a *power-on* message which contains the location of the node and a *random direction*. Each node that does not volunteer *exponentially increases* its volunteering probability when its backoff timer expires, unless it receives a power-on message. This ensures that unconnected nodes eventually become active in the round.

A node's volunteering process ends when the node receives its first power-on message (or becomes active as a starting node). Each time a 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. If not covered, the node checks the following conditions based on the information about its *neighbors*[4]:

a) There exists an uncovered crossing within the coverage area of the node.

b) There are no uncovered crossings, but at least one of its neighbors is a starting node.

c) There are no uncovered crossings or starting neighbors.

In conditions *a* and *b*, the node sets its backoff timer depending on how close the node is to the optimal position. If the node is located at the appropriate distance and in the right direction, the backoff timer

[4]The word *neighbor* has the following meaning: node B is node A's neighbor if A has received a power-on message from B in the current round.

is set to a small value. If not, the 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 optimal position in case b is located at distance $\sqrt{3}$ from the neighbor and in the direction determined by the direction field in the received power-on message (see Fig 2(b).) In case a, the optimal position is located on the midpoint between the two nodes that create the uncovered crossing at a distance to the uncovered crossing equal to the sensing range (see Fig 2(a).) In condition c, the node's backoff timer is set to a large value T_c because "when a node receives only power-on messages from non-starting neighbors, it expects to receive another power-on message and the coverage areas of the two senders will overlap."

The network enters the steady state phase when each node is either active or inactive. When a round is over, each node resets its status to "undecided," and the density control process starts over again.

3 Real-Time Maude

Real-Time Maude [27, 28] is a language and tool extending Maude [9, 10] to support the formal specification and analysis of *real-time* and *hybrid* systems. The specification formalism is based on *real-time rewrite theories* [26]—an extension of *rewriting logic* [6, 20]—and is particularly suitable to specify distributed real-time systems in an object-oriented style. Real-Time Maude achieves high performance by exploiting as much as possible the underlying Maude engine.

Real-Time Maude specifications are *executable* under reasonable assumptions, so that a first form of formal analysis consists in simulating the system's progress in time by *timed rewriting*. This can be very useful for debugging the specification; but of course, any such execution gives us only *one* behavior among the many possible concurrent behaviors of the systems. To gain further assurance about a system design one can use *model checking* techniques that explore many different behaviors from a given initial state of the system. Timed *search* and *time-bounded linear temporal logic model checking* can analyze *all* behaviors (possibly relative to a chosen time sampling strategy, in case we have a dense time domain) from a given initial state up to a certain duration. By restricting search and model checking to behaviors up to a given duration, the set of reachable states can often be restricted to a finite set, which can then be subjected to model checking.

Real-Time Maude offers an alternative to informal specifications and their testing on simulation tools and testbeds by:

- providing a precise formal specification of the system which, being executable, can be simulated and tested directly;
- allowing the specification to be analyzed in many different ways, not just by simulating a few behaviors of the system, but by exhaustively exploring a wide range of different scenarios; and
- allowing the user to define the appropriate forms of communication at a high level of abstraction, instead of having to use a fixed set of communication primitives.

On the other side of the spectrum, Real-Time Maude complements *formal* tools such as the timed/hybrid automaton-based tools Kronos [35], UPPAAL [4], and HyTech [16] by providing a more general specification formalism which supports well the specification and analysis of "infinite-state" systems with different communication and interaction models and with advanced object-oriented and modularity features. Such systems usually fall outside the decidable fragments supported by the aforementioned tools. Finally, some tools geared toward modeling and analyzing larger real-time systems, such as, e.g., IF [5], extend timed automaton techniques with explicit UML-inspired constructions for modeling objects, communication, and some notion of data types. Real-Time Maude complements such tools not only by the full generality of the specification language, but, most importantly, by its simplicity and clarity: A simple and intuitive formalism is used to specify both the data types (by *equations*) and dynamic and real-time behavior of the system (by *rewrite rules*). Furthermore, the operational semantics of a Real-Time Maude specification is clear and easy to understand.

3.1 Preliminaries: Object-Oriented Specification in Maude

Since Real-Time Maude specifications extend Maude specifications, we first recall object-oriented specification in Maude. A Maude module specifies a *rewrite theory* of the form $(\Sigma, E \cup A, \phi, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic* [22] theory with Σ a signature, E a set of conditional equations and memberships, and A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A. The theory $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type. ϕ is a function which associates to each function symbol $f \in \Sigma$ its *frozen*[5] argument positions [10], and R is a collection of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form

$$[l] : t \longrightarrow t' \text{ if } \bigwedge_{i=1}^{n} u_i \longrightarrow v_i \wedge \bigwedge_{j=1}^{m} w_j = w'_j,$$

where l is a *label*. Intuitively, such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t', *provided* the condition holds. The rewrite rules are applied *modulo* the equations $E \cup A$.[6]

We briefly summarize the syntax of Maude. *Functional* modules and *system* modules are, respectively, equational theories and rewrite theories, and are declared with respective syntax `fmod ... endfm` and `mod ... endm`. *Object-oriented* modules provide special syntax to specify concurrent object-oriented systems, but are entirely reducible to system modules; they are declared with the syntax `(omod ... endom)`.[7] Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts* and *subsorts* and *operators*. Operators are introduced with the `op` keyword. They can have user-definable syntax, with underbars '`_`' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. There are three kinds of logical statements, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—*memberships*—declaring that a term has a certain sort and introduced with the keywords `mb` and `cmb`—and *rewrite rules*—introduced with the keywords `rl` and `crl`. The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they must be have the form *var*`:`*sort*. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

In object-oriented Maude modules one can declare *classes* and *subclasses*. A class declaration

`class` C `|` att_1 `:` s_1`,` `...` `,` att_n `:` s_n `.`

declares an object 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

$$\texttt{<}\ O : C \mid att_1 : val_1, ..., att_n : val_n\ \texttt{>}$$

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. Objects can interact with each other in a variety of ways, including the sending of messages. A message is a term of the built-in sort `Msg`, where the declaration

`msg` m `:` p_1 ... p_n `->` `Msg`

defines the syntax of the message (m) and the sorts (p_1 ... p_n) of its parameters. 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 and having the `none` multiset as its identity element, so that order and parentheses

[5] Rewrites cannot take place in a frozen argument position of a function symbol, so that a term $f(t_1, \ldots, t_i, \ldots, t_n)$ will *not* rewrite to $f(t_1, \ldots, u_i, \ldots, t_n)$ when t_i rewrites to u_i if $i \in \phi(f)$.

[6] Operationally, a term is reduced to its E-normal form modulo A before any rewrite rule is applied in Maude.

[7] In Real-Time Maude, being an extension of Full Maude, module declarations and execution commands must be enclosed by a pair of parentheses.

do not matter, and so that rewriting is *multiset rewriting* supported directly in 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 configuration fragment on the left-hand side of the rule

```
rl [l] :  m(O,w)  < O : C | a1 : x, a2 : y, a3 : z >  =>
                  < O : C | a1 : x + w, a2 : y, a3 : z >  m'(y,x)
```

contains a message `m`, with parameters `O` and `w`, and an object `O` of class `C`. The message `m(O,w)` does not occur in the right-hand side of this rule, and can be considered to have been *removed* from the state by the rule. Likewise, the message `m'(y,x)` only occurs in the configuration on the right-hand side of the rule, and is thus *generated* by the rule. The above rule, therefore, defines a parameterized family of transitions in which a message `m(O,w)` is read and consumed by an object `O` of class `C`, with the effect of altering the attribute `a1` of the object and of sending a new message `m'(y,x)`. By convention, attributes, such as `a3` in our example, whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule. Attributes like `a2` whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, may be omitted from right-hand sides of rules.

3.2 Object-Oriented Specification in Real-Time Maude

A Real-Time Maude *timed module* (syntax `(tmod ... endtm)`) specifies a *real-time rewrite theory* [26, 28], that is, a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$, such that:

1. $(\Sigma, E \cup A)$ contains an equational subtheory $(\Sigma_{TIME}, E_{TIME}) \subseteq (\Sigma, E \cup A)$, satisfying the $TIME$ axioms in [26], which specifies a sort `Time` as the time domain (which may be discrete or dense). Real-Time Maude provides some predefined modules specifying useful time domains. For example, the modules `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF` define the time domain to be, respectively, the natural numbers and the nonnegative rational numbers. These modules also add a supersort `TimeInf`, which extends the sort `Time` with an "infinity" value `INF`.

2. The sort of the "states" of the system has the designated sort `System`.

3. The rules in R are decomposed into:

   - "ordinary" rewrite rules that model *instantaneous* change, and
   - *tick (rewrite) rules* that model the elapse of time in a system. Such tick rules must be of the form $l : \{t\} \longrightarrow \{t'\}$ **if** *cond*, where t and t' are of sort `System`, $\{\_\}$ is a built-in constructor of a new sort `GlobalSystem`, and where we have associated to such a rule a term u of sort `Time` denoting the *duration* of the rewrite. In Real-Time Maude, tick rules, together with their durations, are specified with the syntax

     `crl [`l`] : {`t`} => {`t'`} in time` u `if` *cond* `.`

All ground terms of sort `GlobalSystem` must be reducible to terms of the form `{`t`}` using the equations in the specification. The form of the tick rules then ensures uniform time elapse in all parts of a system.

Timed object-oriented modules (syntax `(tomod ... endtom)`) extend both object-oriented and timed modules to provide support for object-oriented specification of real-time systems. Timed object-oriented modules include built-in subsorts such as `NEConfiguration` for non-empty configurations. The sort `Configuration` is declared to be a subsort of the sort `System`.

3.3 Rapid Prototyping and Formal Analysis in Real-Time Maude

We summarize below the Real-Time Maude analysis commands used in our case study. All Real-Time Maude analysis commands are described in [30], and their mathematical semantics is given in [28]. Note that all analyses are performed with respect to the chosen *time sampling strategy* treatment of the tick rule(s) [27, 28].

3.3.1 Rapid Prototyping: Timed Rewriting

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 <=` *timeLimit* `.)`

where t is the term to be rewritten ("the initial state"), and *timeLimit* is a ground term of sort `Time`. Our tool also provides facilities for *tracing* the rewrite steps performed in a simulation (see [30]).

3.3.2 Search and Model Checking

Real-Time Maude provides a variety of search and model checking commands for further analyzing timed modules by exploring *all* possible behaviors—up to a given number of rewrite steps, duration, or satisfaction of other conditions—that can be nondeterministically reached from the initial state.

First of all, Real-Time Maude extends Maude's *search* command—which uses a breadth-first strategy to search for states that are reachable from the initial state which match the *search pattern* and satisfy the *search condition*—to search for states which can be reached within a given time interval from the initial state. The search command has syntax

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

where t is the initial state, *pattern* is the search pattern, *cond* is a semantic condition on the variables in the search pattern, and *timeLimit* is a ground term of sort `Time`. The command then returns at most n states that are solutions of the search. The `such that`-condition may be omitted.

Real-Time Maude provides commands for analyzing all behaviors from the initial state by searching for the *earliest* and the *latest* time when a certain state is reached for the first time. The command

`(find earliest` t `=>*` *pattern* `such that` *cond* `.)`

finds the earliest state reachable from t which is matched by *pattern* and satisfies *cond*. The command

`(find latest` t `=>*` *pattern* `such that` *cond* `in time <=` *timeLimit* `.)`

searches through all behaviors, and finds the *first* occurrence of a *pattern*-state satisfying *cond* in each behavior. Among these states, the state which took the longest time to reach is returned. The execution of this command will return "not found in all computations" if there is a behavior in which the desired state cannot be reached within the time limit.

Finally, Real-Time Maude extends Maude's *linear temporal logic model checker* [13, 10] to check whether each behavior "up to a certain time," as explained in [28], satisfies a temporal logic formula. Restricting the computations to their time-bounded prefixes means that properties can be model checked in specifications that do not allow *Zeno behavior*, since only a finite set of states can then be reached from an initial state. *State propositions*, possibly parameterized, should be declared as operators 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 $\{t\}$ such that $\{t\}$ `|=` *prop* evaluates to `true`. It is not necessary to define explicitly the states in which *prop* does not hold. A temporal logic *formula* is constructed by state and clocked propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `W` ("weak until"). The command

`(mc` t `|=t` $formula$ *timeLimit* `.)`

is the timed model checking command which checks whether the temporal logic formula $formula$ holds in all behaviors up to duration *timeLimit* starting from the initial state t.

4 The Real-Time Maude Specification of the OGDC Algorithm

This section presents a sample of our Real-Time Maude specification of the OGDC algorithm.[8] Section 4.1 explains how locations and sensing areas are represented in the model, and defines some functions on such entities. Section 4.2 shows how time and time elapse are modeled. Section 4.3 defines the sensor nodes. Section 4.4 explains how we model communication in wireless sensor networks. In Section 4.5 we introduce a pseudo-random number generator which is used to simulate probabilistic features. Finally, Section 4.6 presents some of the rewrite rules that define the dynamic behavior of the OGDC algorithm.

4.1 Geometric Computations

This section defines data types for coverage areas, distances, and angles. We assume that the sensor nodes are located on a two dimensional surface and represent a *location* as a pair of rational numbers of sort `Location`:

```
sort Location .
op _._ : Rat Rat -> Location [ctor] .
```

For example, the term `(45 . 3/2)` denotes the location 45 centimeters along the x-axis and 3/2 centimeters along the y-axis in a fixed coordinate system.

The function `vectorLengthSq` computes the distance *squared*[9] between two locations, and the function `withinSensingRangeOf` checks whether or not two locations are within sensing range of each other:

```
op vectorLengthSq         : Location Location -> Rat .
op _withinSensingRangeOf_ : Location Location -> Bool .

vars L L' : Location .    vars X X' Y Y' : Rat .

eq vectorLengthSq(X . Y, X' . Y') = ((X - X') * (X - X')) + ((Y - Y') * (Y - Y')) .

eq L withinSensingRangeOf L' = vectorLengthSq(L, L') <= (sensingRange * sensingRange) .
```

In order to set the backoff timers we need to compute different angles. We define a function `angle` which computes the angle between a vector, defined by two locations, and the x-axis.[10]

```
op angle : Location Location -> Rat .

ceq angle(X . Y, X' . Y') =
      (if negY(L) then 2 * pi - acos(dotProd(L, 1 . 0))
       else acos(dotProd(L, 1 . 0)) fi)
  if L := normalize((X' - X) . (Y' - Y)) .
```

The angle between two vectors is found by computing the inverse cosine of the dot product of the two normalized vectors. The function `normalize` normalizes a vector, and `dotProd` computes the dot product of two vectors. The function `negY` checks whether the second coordinate of the vector is negative (that is, if the angle between the vector and the x-axis is larger than π). If this is the case, the conjugate[11] angle is computed.

4.1.1 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 [37] to use a bitmap to model

[8] Our specification is explained in greater detail in [34]. The entire executable Real-Time Maude specification can be found at `http://www.ifi.uio.no/RealTimeMaude/OGDC`.

[9] We compute with *squares* to avoid square roots as much as possible.

[10] We have extended the operators `pi` and `acos` from the sort `Float` to the sort `Rat` by definitions of the form `acos(R) = rat(acos(float(R)))` using Maude's conversion operators `rat` and `float`. The resulting rounding inaccuracy can be tolerated in the algorithm.

[11] Two angles are called conjugate if they add up to 2π.

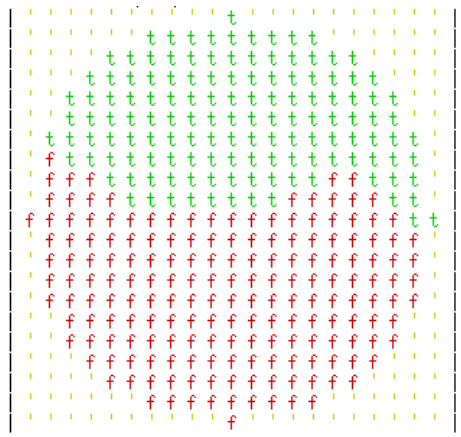


Figure 3: The bitmap for a node's coverage area.

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 Maude tool is not a graphical tool, but with proper use of the `format` operator attribute [10], a bitmap can be given an intuitive appearance as shown in Fig. 3. We define a bitmap as a term of sort `Bitmap`, which consists of a list of `BitLists`[12], which in itself is a list of `Bits`:

```
sorts Bitmap BitList Bit .    subsort Bit < BitList .
```

A term of sort `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. 3. The bits are declared with appropriate colors:

```
op t : -> Bit [ctor format (g o)] .
op f : -> Bit [ctor format (r o)] .
op ' : -> Bit [ctor format (y o)] .
```

The `Bits` are concatenated into a `BitList` as follows:

```
op nil : -> BitList [ctor] .
op __  : BitList BitList -> BitList [ctor assoc id: nil format (o s o)] .
```

The `format` attribute causes a blank to be inserted between each bit. We enclose a `BitList` by a `|...|` operator so that we can insert a new line before each `BitList` with the `format` attribute:

```
op |_| : BitList -> Bitmap [ctor format (ni o o o)] .
```

The enclosed `BitLists` are finally concatenated into a `Bitmap`:

```
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. All the bits in the bitmap that are within the sensing range of the node, and within the sensing area of the system, are initialized to `f`. The bits outside the sensing range and the sensing area are initialized to `'`.

A function `updateBitmap` is used to update a node's bitmap each time the node receives a power-on message. This function traverses the bitmap and checks, using the `withinSensingRangeOf` function defined above, and changes each bit that has value `f` to `t` if it is within the sensing range of the sender of the power-on message.

Each time a node receives a power-on message, the node also checks whether its bitmap (updated with the sender of this power-on message) is completely covered by its neighbors. This is done by checking the value of each bit in the bitmap with the function `coverageAreaCovered` as follows:

[12] Each `BitList` corresponds to a "row" in the bitmap.

```
var BIT : Bit .    var BITL : BitList .   var BM : Bitmap .

 op coverageAreaCovered : Bitmap -> Bool .
 eq coverageAreaCovered(nil) = true .
 eq coverageAreaCovered(| nil | BM) = coverageAreaCovered(BM) .
 eq coverageAreaCovered(| BIT BITL | BM) =
        (BIT =/= f) and coverageAreaCovered(| BITL | BM) .
```

Since a node's bitmap is often traversed when it receives a power-on message, the number of bits in the bitmap has a significant impact on the execution times of the system. We use a distance of 100 centimeters between each bit in a node's bitmap, which results in 400 bits (including the ' bits used for padding) since the sensing range of a node is 1000 centimeters. Simulations of the OGDC algorithm for 400 nodes for one round using a bitmap of this size takes about 45 minutes to execute on a 2GHz Pentium Xeon processor.

4.1.2 Coverage Area Crossings

When a node receives a power-on message, it needs to compute the crossings that the new neighbor's coverage area creates with the coverage areas of the node's existing neighbors. A crossing is represented in the model by the location of the two nodes that create the crossing, and the location of the crossing:

```
sorts Crossing CrossingSet .    subsort Crossing < CrossingSet .

op _x_in_ : Location Location Location -> Crossing [ctor] .
op none   : -> CrossingSet [ctor] .
op __     : CrossingSet CrossingSet -> CrossingSet [ctor assoc comm id: none] .
```

Each node maintains a set of uncovered crossings in its coverage area in order to efficiently compute its backoff timer value. The function `updateUncoveredCrossings` updates a set of crossings, created by a given set of nodes, with an additional node:

```
op updateUncoveredCrossings : Location Location NeighborSet CrossingSet -> CrossingSet .
```

The formulas we use for computing the locations of the crossings between two nodes were found in a preliminary version of [37].

4.1.3 Computing the Backoff Timer Values

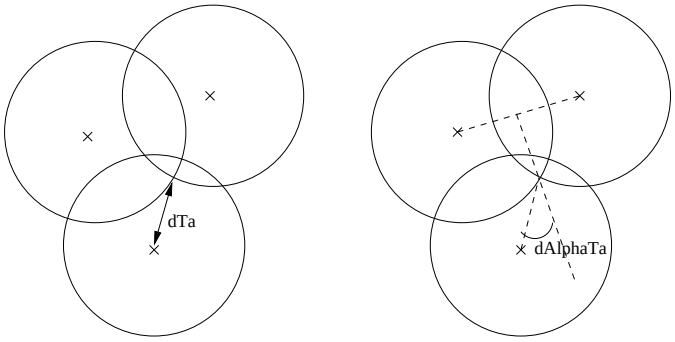
When an undecided node receives a power-on message, the node computes its backoff timer value depending on the conditions given in Section 2.1. In condition *a*, the node computes a new backoff timer value, T_a, if the sender of the power-on message creates the *closest* uncovered crossing. The backoff timer values are computed from formulas given in [37]. The formula for computing T_a is given by

$$T_a = t_0(c((r_s - d)^2 + (d\Delta\alpha)^2) + u)$$

where t_0 is the transmission delay, c is a constant that determines the backoff timer scale, r_s is the sensing range, d is the distance between the receiving node and the crossing (see Fig. 4(a)), $\Delta\alpha$ is the angle between the optimal position, with respect to the crossing, and the receiving node (see Fig. 4(b)), and u is a random term uniformly distributed on $[0, 1]$. The resulting backoff timer value T_a, according to [37], "roughly represents the deviation from the optimal position." That is, the closer the node is to the optimal position (see Fig 2(a)), the smaller its backoff timer value becomes. Therefore, the node that is closest to the optimal position will become active first. We define a function `setTa` that computes the value T_a:

```
op setTa : Oid Nat CrossingSet -> Time .
var O : Oid .    var N : Nat .    var CS : CrossingSet .    var C : Crossing

ceq setTa(O, N, CS) =
             ceiling(transmissionDelay *
```



(a) The distance between the receiving node and the closest crossing.

(b) The angle between the optimal position with respect to closest uncovered crossing and the receiving node.

Figure 4: Distance and angle in the computation of the backoff timer value T_a.

```
                    (c * (((sensingRange - dTa(O, C)) * (sensingRange - dTa(O, C)))
                           + (dTa(O, C) * dTa(O, C) * dAlphaTa(O, C) * dAlphaTa(O, C)))
                     + randomU(N)))
  if C:= closestCrossing(O, CS) .
```

See [34] for the complete definition of the functions `dTa`, `dAlphaTa`, and the functions computing the backoff timer values T_b and T_c.

4.2 Modeling Time and Time Elapse

We follow the guidelines suggested in [25, 29] for modeling time-dependent behavior in object-oriented specifications. A function `delta` is used to define the effect of time elapse on objects and messages in a configuration, and a 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 configuration as follows:

```
vars NEC NEC' : NEConfiguration .    var T : Time .

op delta : Configuration Time -> Configuration [frozen (1)] .
eq delta(none, T) = none .
eq delta(NEC NEC', T) = delta(NEC, T) delta(NEC', T) .

op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(none) = INF .
eq mte(NEC NEC') = min(mte(NEC), mte(NEC')) .
```

The functions `delta` and `mte` must then be defined for single objects and messages as described in Sections 4.3 and 4.4. The "standard" tick rule of Ölveczky and Meseguer is used to model time elapse:

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

This tick rule advances time nondeterministically by *any* amount `T` less than or equal to `mte(C)`. The concrete value of `T` is not given until a *time sampling strategy* is chosen (in Section 5).

Real-Time Maude supports both discrete and dense time. Our specification is essentially parametric in the time domain. Since we did not find any compelling reason to assume dense time in the OGDC algorithm, we defined the time domain to be the natural numbers by importing 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` (denoting ∞) of a supersort `TimeInf`.

4.3 The Definition of Sensor Node Objects

We model nodes in the wireless sensor network as objects of the class `WSNode`. Since we have assumed that a localization protocol has been used prior to the start of the OGDC algorithm, each sensor node is aware of its geographical location. We use the location of a node as its unique *identifier* by declaring the sort `Location` to be a subsort of the sort `Oid`. The class `WSNode` is declared as follows:

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

The attributes of this class denote the following:

`backoffTimer`: The time remaining until the node must perform an action.

`bitmap`: Denotes what sections of the node's coverage area are covered by its neighbors.

`uncoveredCrossings`: The set of uncovered crossings within the node's coverage area.

`neighbors`: The set of the node's neighbors, that is, the nodes from which the node has received a power-on message in the current round.

`remainingPower`: The amount of power that the node has left.

`roundTimer`: The amount of time remaining until the next round of OGDC starts.

`status`: The node's status: `on`, `off`, or `undecided`.

`volunteerProb`: The probability for the node to volunteer as a starting node.

`hasVolunteered`: Denotes whether the node volunteered as a starting node. Initialized to `undecided`, changed to either `true` or `false` after its volunteering process.

A `Neighbor` in the `NeighborSet` is represented by the location of the neighbor and by a Boolean value which is `true` when the neighbor is a starting node. A starting neighbor at location `(2/5 . 65)` is represented by the term `(2/5 . 65 starting true)` in the attribute `neighbors`.

Timed Behavior of a Node

The function `delta` defines the effect of time elapse on a `WSNode` object by decreasing its backoff timer and round timer by the amount of time that has elapsed. In addition, the amount of remaining power must be decreased according to the elapsed time and whether the node is turned on or off:

```
var O : Oid .     var T : Time .   vars TI TI' : TimeInf .     var P : NzNat .
var S : Status .  var N : Nat .    var V : VolunteeredStatus .

eq delta(< O : WSNode | remainingPower : N, status : S,
                        backoffTimer : TI, roundTimer : TI' >, T)
       =
         < O : 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 built-in function `monus` is defined on the time domain (including on the additional value `INF`) by x `monus` $y = x - y$ if $x \geq y$, and `0` otherwise.

We define the function `mte` on sensor nodes so that time cannot elapse when a node is in its volunteering process, i.e., when its `hasVolunteered` attribute is set to `undecided`:

```
eq mte(< O : WSNode | hasVolunteered : undecided >) = 0 .
```

When a node has exhausted its power supply, it should die (i.e., set its `status` to `off`) immediately. A "dead" node should not put any constraint on the amount time can elapse:

```
eq mte(< O : WSNode | remainingPower : 0, status : S >) =
     if S == off then INF else 0 fi .
```

Time should not advance beyond the expiration time of either the backoff timer or the round timer of a node that is alive. Furthermore, to ensure a timely death of a sensor node, time should not advance beyond the time left until the node is out of power (`P / powerUnit`):

```
ceq mte(< O : WSNode | backoffTimer : TI, roundTimer : TI', remainingPower : P,
                       hasVolunteered : V, status : S >) =
    min(TI, TI', if S == on then ceiling(P / powerUnit) else P fi)
 if V =/= undecided .
```

4.4 Modeling Communication in Wireless Sensor Networks

Sensor nodes equipped with *(undirected) radio transmitters* communicate by *broadcast*. Since the transmitters are fairly weak, the range of the broadcast signal is limited, which means that only sensor nodes within a certain geographical distance from the sender will receive the signal with sufficient strength. The wireless communication capacity is assumed to be 40 Kbs, while the packet size for the *power-on* packets is 34 bytes [37]. Communication is therefore subject to *transmission delays* which should be modeled. It is worth remarking that potential media access problems (two nodes broadcasting simultaneously) are not considered at the abstraction level of the OGDC algorithm description.

Wireless communication pose some challenges to their formal modeling:

- The sender does not know the other nodes (within transmission range) in the system.[13] Multicast techniques are therefore not well suited to model this kind of broadcast.
- The broadcast packet must reach *all* nodes that are within a certain *geographical distance* from the sender.
- The packets are subject to transmission delays.

Real-Time Maude provides a flexible formalism where domain-specific communication models can be defined. The main idea behind our communication model is that the sender sends a *broadcast* message into the configuration. This broadcast message is defined to be *equivalent* to a set of single, addressed messages, each of which is delivered after the transmission delay has expired.

Modeling message delay for single messages is as suggested in [30, 29], namely by introducing a delay operator `dly`, which is declared as follows:

```
sort DlyMsg .  subsorts Msg < DlyMsg < NEConfiguration .
op dly : Msg TimeInf -> DlyMsg [ctor right id: 0] .
```

The idea is that a message `dly(`m`,`t`)` denotes that the message m will be "ready" in time t. If it must be received *exactly* in time t, we can define `delta` and `mte` on delayed messages as follows:

```
var M : Msg .   var T : Time .  var TI : TimeInf .
eq mte(dly(M, TI)) = TI .
eq delta(dly(M, TI), T) = dly(M, TI monus T) .
```

In our model, a sensor node broadcasts a *power-on* packet[14] by sending a broadcast message to the configuration. Single power-on messages and the broadcast message are declared as follows:

```
msg broadcast'from_withDirection_ : Oid Int -> Msg .
msg power-onMsgFrom_to_withDirection_ : Oid Oid Int -> Msg .
```

[13] A node only know its *active* neighbors *after* the node selection phase in each round.
[14] *Power-on* packets are the only kind of packets broadcast in this algorithm.

(The "withDirection" field does *not* mean that the broadcast is sent in a certain direction; it is a parameter of the power-on packet as explained in Section 2.) A rule modeling a node broadcasting a power-on packet should have the form

```
crl [l] :
    ...  < O : WSNode | ... >
   =>
         < O : WSNode | ... >  broadcast from O withDirection d .
```

The idea is that a "broadcast message" is equivalent to a set of single, addressed messages; one to each node that is within the transmission range of the sender. The following equation captures the desired equivalence:

```
eq {< O : WSNode | > (broadcast from O withDirection D) C} =
   {< O : WSNode | > distributeMsg(O, D, C)} .
```

It is the task of distributeMsg to create an addressed power-on messages for each sensor object in C that is located within the transmission range of O. The use of the operator {_} enables the equation to grab the *entire* state to make sure that *all* appropriate nodes in the system will get the message. The function distributeMsg is defined as follows:

```
op distributeMsg : Oid Nat Configuration -> Configuration [frozen (3)] .

eq distributeMsg(O, D, none) = none .
eq distributeMsg(O, D, MSG C) = MSG distributeMsg(O, D, C) .
eq distributeMsg(O, D, < Random : RandomNGen | > C) =
    < Random : RandomNGen | > distributeMsg(O, D, C) .

eq distributeMsg(O, D, < O' : WSNode | > C) =
    < O' : WSNode | >  distributeMsg(O, D, C)
    if O withinTransmissionRangeOf O'
    then dly((power-onMsgFrom O to O' withDirection D), transmissionDelay)
    else none
    fi .
```

The function withinTransmissionRangeOf is defined as expected:

```
eq O withinTransmissionRangeOf O' =
       vectorLengthSq(O, O') <= (transmissionRange * transmissionRange) .
```

In this model, broadcasting and receiving messages can be done by rewrite rules in the usual Maude style explained in Section 3.1.

4.5 Random Number Generation

We simulate probabilistic aspects of the OGDC algorithm by using the following function random, which generates a sequence of numbers pseudo-randomly and satisfies the criteria of a "good" random function given in [18]:

```
op random : Nat -> Nat .
eq random(N) = ((104 * N) + 7921 .
```

A new class RandomNGen with an attribute seed is used to store the ever-changing "seed" to random in the state.

4.6 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. The commonly used variables in the rules are:

```
var M : Nat .        var D : Int .        var R : Rat .          var P : NzNat .
vars O O' : Oid .    var BM : Bitmap .    var NBS : NeighborSet .
var S : Status .     var T : Time .       var CS : CrossingSet .
```

4.6.1 Selection of the Starting Nodes

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 [37]:

> *At the beginning of node selection phase, every node is powered on with the "UNDECIDED" state. 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. If a node hears other power-on messages before its timer expires, it cancels its timer and does not become a starting node. 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 $[0, 2\pi]$. Non-starting node may also send power-on message. In this case, the direction field in the power-on message is set to -1 to indicate the sender is a non-starting node. [...] If the node does not volunteer itself to be a starting node, it sets a timer of T_s seconds. When the timer T_s expires, it repeats the above volunteering process with p doubled until its value reaches 1. The timer is canceled whenever the state of a node is changed to "ON" or "OFF" in response to other power-on messages.*

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`:

```
rl [volunteer] :
  < O : WSNode | remainingPower : P, volunteerProb : R, hasVolunteered : undecided >
  < Random : RandomNGen | seed : M >
=>
  (if (randomProb(M) < R) and (P > powerThreshold or R == 1000)
   then < O : WSNode | backoffTimer : randomTimer(random(M)), hasVolunteered : true >
   else < O : WSNode | backoffTimer : nonVolunteerTimer, hasVolunteered : false,
                       volunteerProb : doubleProb(R) >
   fi)
  < Random : RandomNGen | seed : repeatRandom(M, 3) > .
```

The node must also have sufficient remaining power (`P > 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 10 (T_d) by the `randomTimer` function. If the node does not volunteer, it sets its backoff timer to a constant `nonVolunteerTimer` (T_s).

The following rewrite rule models the case where the backoff timer of a non-volunteered node expires (that is, reaches the value `0`) without the node having received a single power-on message (its `neighbors` set is still `none`). The node repeats the volunteering process with the probability for volunteering doubled:

```
rl [volunteer2] :
   < O : WSNode | backoffTimer : 0, neighbors : none, remainingPower : P,
                  volunteerProb : R, hasVolunteered : false >
   < Random : RandomNGen | seed : M >
 =>
   (if (randomProb(M) < R) and (P > powerThreshold or R == 1000)
    then < O : WSNode | backoffTimer : randomTimer(random(M)), hasVolunteered : true >
    else < O : WSNode | backoffTimer : nonVolunteerTimer, volunteerProb : doubleProb(R) >
    fi)
  < Random : RandomNGen | seed : repeatRandom(M, 3) > .
```

A node becomes *active* when its backoff timer expires and, in addition, either the node has volunteered or has received at least one power-on message. In the first case, the node becomes active as a starting node and broadcasts a power-on message that contains the node's location and a *random direction*:

```
rl [startingNodePowerOn] :
   < O : WSNode | remainingPower : P, backoffTimer : 0,
                  hasVolunteered : true >
   < Random : RandomNGen | seed : M >
=>
   < O : WSNode | remainingPower : P monus transPower,
                  backoffTimer : INF, status : on >
   < Random : RandomNGen | seed : random(M) >
   broadcast from O withDirection randomDirection(M) .
```

The node consumes `transPower` amount of power when it broadcasts a power-on message. The rule `nonStartingNodeSwitchOn` for a non-volunteered node becoming active[15] is done similarly but the value `-1` is put in the direction field of the broadcast message:

```
rl [nonStartingNodePowerOn] :
   < O : WSNode | remainingPower : P, backoffTimer : 0,
                  neighbors : NB NBS, hasVolunteered : false >
  =>
   < O : WSNode | remainingPower : P monus transPower,
                  backoffTimer : INF, status : on >
   broadcast from O withDirection -1 .
```

4.6.2 Receiving a Power-On Message

The following three rules model the reception of a power-on message when the delay "timer" of the power-on message has expired. The actions taken when a node receives a power-on are described as follows in [37][16]:

> *When a sensor node receives a power-on message, if the node is already "ON", or it is more than* $2\,r_s$ *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 . . .*

When the receiving node is in state `on` or `off`, or the distance between the sender and the receiver is greater than $2\,r_s$, the power-on message is just ignored:

```
crl [discard] :
    (power-onMsgFrom O' to O withDirection D)
    < O : WSNode | status : S >
  =>
    < O : WSNode | >
  if S =/= undecided or not (O withinTwiceTheSensingRangeOf O') .
```

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 [recPowerOnMsgAndSwitchOff] :
    (power-onMsgFrom O' to O withDirection D)
    < O : WSNode | status : undecided, neighbors : NBS, bitmap : BM >
  =>
    < O : WSNode | status : off, neighbors : NBS (O' starting (D >= 0)),
                   bitmap : updateBitmap(O, BM, O'),
                   backoffTimer : INF >
  if (O withinTwiceTheSensingRangeOf O')
     /\ sensingAreaCovered(updateBitmap(O, BM, O')) .
```

[15] A node will only match this rule if its `neighbors` set is non-empty.
[16] We only quote the first part of that description.

An `undecided` node that does not get its entire coverage area covered, receives the power-on message with one of three rules, corresponding to the three conditions *a*, *b*, and *c* listed in Section 2.1. All three rules add the sender of the power-on message to the receiving node's set of neighbors and update the bitmap of the receiver.

If a node receives a power-on message and has at least one uncovered crossing within its coverage area (`updateUncoveredCrossings(...) =/= none`), it (re)sets its backoff timer to T_a (`setTa(...)`) if the sender of the latest power-on message creates the *closest* uncovered crossing:

```
crl [recPowerOnWithUncoveredCrossings] :
    (power-onMsgFrom O' to O withDirection D)
    < O : WSNode | status : undecided, backoffTimer : T,
                   neighbors : NBS, uncoveredCrossings : CS, bitmap : BM >
    < Random : RandomNGen | seed : M >
 =>
    < O : WSNode | backoffTimer : (if O' createsClosestCrossing
                                          O (updateUncoveredCrossings(O, O', NBS, CS))
                                    then setTa(O, M, updateUncoveredCrossings(O, O', NBS, CS))
                                    else T
                                    fi),
                   neighbors : NBS (O' starting (D >= 0)),
                   uncoveredCrossings : updateUncoveredCrossings(O, O', NBS, CS),
                   bitmap : updateBitmap(O, BM, O') >
    < Random : RandomNGen | seed : random(M) >
  if (O withinTwiceTheSensingRangeOf O')
     /\ updateUncoveredCrossings(O, O', NBS, CS) =/= none
     /\ not coverageAreaCovered(updateBitmap(O, BM, O')) .
```

The other two rules corresponding to conditions *b* and *c* in Section 2.1 are given similarly.

4.6.3 Other Actions

The death of a node is modeled by an equation that simply changes the node's status attribute to `off`. The definition of `mte` that stops time elapse when a node is out of power, secures an instant death for "powerless" nodes.

When the round is over, a rule `restart` is applied to reset the attributes, except the remaining power, of each living node to their initial values:

```
rl [newRound] :
   < O : WSNode | roundTimer : 0, remainingPower : P >
 =>
   < O : WSNode | status : undecided, neighbors : none, uncoveredCrossings : none,
                  bitmap : initBitmap(O), hasVolunteered : undecided, backoffTimer : INF,
                  roundTimer : roundTime, volunteerProb : 1000 / n > .
```

5 Analysis of the OGDC Algorithm

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*, where probabilistic behavior is simulated using our pseudo-random number generator, using *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 in a given sensor network topology, but only those behaviors that can take place with the specific choice of pseudo-random numbers used to simulate the probabilistic behavior.

In Section 5.1, we define the *time sampling strategy* which defines how the nondeterministic tick rewrite rule should be applied, and show how we can easily generate initial states with a large number of sensor nodes scattered pseudo-randomly in a given sensing area. Section 5.2 shows how simulations corresponding to those performed in [37] with the ns-2 tool can be done in Real-Time Maude. Section 5.3 contains examples of how timed search and LTL model checking can be used to perform exhaustive state space exploration.

5.1 Defining Initial States and the Time Sampling Strategy

We use a function `genInitConf` to conventiently generate initial states. The term `genInitConf(M, N)` defines an initial configuration[17] 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 `M`.[18] We can therefore easily generate initial states with any number of nodes, and place them in different locations, by just changing the parameters `M` and `N` in `genInitConf`. The function `genInitConf` is defined as follows:

```
op genInitConf : Nat Nat -> Configuration .       --- seed numNodes
op genInitConf : Nat Nat Nat -> Configuration .
eq genInitConf(M, N) = genInitConf(M, N, N) .

ceq genInitConf(M, s(N), N') =
      < L : WSNode | remainingPower : lifetime, status : undecided,
                     neighbors : none, bitmap : initBitmap(L),
                     uncoveredCrossings : none, backoffTimer : INF,
                     roundTimer : roundTime, volunteerProb : 1000 / N',
                     hasVolunteered : undecided >
      (if N == 0 then < Random : RandomNGen | seed : repeatRandom(M, 3) >
       else genInitConf(repeatRandom(M, 3), N, N') fi)
    if L := random(M) rem sensingAreaSize - (sensingAreaSize / 2) .
          random(random(M)) rem sensingAreaSize - (sensingAreaSize / 2) .
```

As mentioned in Section 4.2, a time sampling strategy must be chosen before the analysis can take place. When we have a discrete time domain, *all* possible behaviors (again, with respect to our simulation of probabilistic behaviors) starting from a given initial state can be investigated by setting the time sampling strategy to advance time by 1 unit each time. However, since all events in the OGDC algorithm happen at specific times, we can "fast forward" between these events without losing any interesting behaviors. Therefore, in our analysis, we use for efficiency reasons the time sampling strategy declared by the Real-Time Maude command:

```
(set tick max def roundTime .)
```

which advances time as much as possible (defined by `mte`) and is advanced by `roundTime` (the length of one round of the OGDC algorithm) if the maximum possible time increase is infinity (this is the case when all the nodes are dead).

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

In [37], Zhang and Hou use the network simulation tool *The Network Simulator* (ns-2) [23], with the wireless extension developed by the CMU Monarch group [11], to simulate the OGDC algorithm and measure the following essential *performance metrics*:

- The number of *active* nodes and the percentage of coverage provided by those nodes at the end of the first round (Section 5.2.2).
- The percentage of coverage and the total amount of remaining power for the whole system throughout the network's lifetime (Section 5.2.3).

[17] The function `genInitConf` generates terms of sort `Configuration` for reasons that will be apparent later. An initial state must also add the operator `{_}`.

[18] The size of the sensing area is not a parameter of `genInitConf` but is given by the constant `sensingAreaSize`.

- The α-*lifetime* (that is, the total time during which at least α percent of the sensing area is covered) for different values of α, and α-lifetime for different number of deployed nodes. (This analysis can be 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.[19] Therefore, we add to the initial state a new construction, that we call an *analysis message*.[20]

5.2.1 Analysis Messages

We use an *analysis message* to compute a performance metric at the same time (e.g., just before the end of the round) in each round of the OGDC algorithm. The computed values are stored in the analysis message as a list n_1 `++` n_2 `++` $\cdots$ `++` n_k, where n_i denotes the value of the metric in round i. The analysis message remains in the state throughout the execution and can be reviewed afterwards. Given a sort `NatList` of lists of natural numbers, with concatenation operator `_++_` and empty list `nil`, we can declare an analysis message `activeNodes`, which computes the number of active nodes in each round, as an ordinary message:

```
msg activeNodes : NatList -> Msg .
```

In the following rule, the analysis message is ripe (i.e., has no delay). It computes and stores the number of active nodes in the system, and resets its delay in order for the message to be ripe again at the same time in the next round:

```
var SYSTEM : Configuration .  var NL : NatList .
rl [computeNumActiveNodes] :
   {activeNodes(NL)  SYSTEM}
  =>
   {dly(activeNodes(NL ++ numActiveNodes(SYSTEM)), roundTime)  SYSTEM} .
```

The function `numActiveNodes` computes the number of active nodes in a configuration. We have defined the analysis messages `coveragePercentage` and `totalRemainingPower`, which compute, respectively, the percentage of the sensing area covered by the active nodes and the total amount of power in the system, in the same way.

5.2.2 Measuring the Number of Active Nodes and the Percentage of Coverage

The first simulations in [37] investigate how the number of active nodes and the percentage of coverage in the *first* round of the algorithm changes with the number of deployed nodes. They vary the number of deployed nodes from 100 to 1000 in a $50m \times 50m$ sensing area. In Real-Time Maude, we can change the number of nodes by just changing the parameter to `genInitConf`. The following timed fair rewrite command simulates a system with 400 nodes and the same sensing area (given by the constant `sensorAreaSize`) until the end of the first round of the protocol (`in time < roundTime`). The initial state contains the two analysis messages which will be ready, and hence compute their metrics, just before the end of the first round:[21]

```
Maude> (tfrew {genInitConf(1, 400)
               dly(activeNodes(nil), roundTime - 1)
               dly(coveragePercentage(nil), roundTime - 1)}
        in time < roundTime .)
```

[19] In principle, one *could* of course use Real-Time Maude's tracing capabilities to trace the state at these different points in an execution, but this is not practical, given the large states and the large number of rewrites involved.

[20] The use of "message" for analysis messages is a slight abuse of the concept of messages, since the analysis messages are not sent or received by any nodes. They just provide a convenient way of computing different metrics at specific times and "storing" the result in the configuration.

[21] The output of Real-Time Maude executions will be manually tabulated, and parts of the output omitted in the exposition will be replaced by '...'.

```
Result ClockedSystem :
  {dly(activeNodes(44), 1000000)
   dly(coveragePercentage(100), 1000000)
   < 38 . -852 : WSNode | ... >    ... }  in time 999999
```

As shown in the analysis messages, 44 of the 400 deployed nodes became active nodes and together provided 100% coverage of the sensing area. Additional timed rewrite simulations with the same number of nodes but with different initial seeds resulted in an average of about 45 active nodes that always provided 100% coverage. Further simulations resulted in an average of 32 active nodes in states with 200 deployed nodes, and in about 38 active nodes among 300 deployed nodes. All simulations showed 100% coverage. The results from the simulations in [37] show that the number of active nodes in the network is between 15 and 20 nodes, which provide 100% coverage when 500 or more nodes are deployed. When fewer nodes were deployed, 98-99% coverage was provided.

It is a clear tendency that our simulations result in a higher number of active nodes. The obvious first place to look for explanations of this fact is to consider the way the sensor nodes are placed in the sensing area. We use pseudo-random numbers to assign locations to the sensor nodes. We do not know how the nodes are placed in the ns-2 simulations. In contrast to their simulations, we also get more active nodes when more nodes are deployed. One possible explanation of this is the following: When more nodes are deployed, more nodes will be almost equally close to the different "optimal" positions, and will therefore get similar backoff timer values. Indeed, when two nodes are equally close to the optimal position, they will have backoff timer values (defined by T_a) which differ by *less than the transmission delay* of power-on messages (since u in the definition of T_a is a number between 0 and 1). Although this random value u was added to "break ties," such tie-breaking will *not* be achieved, since the larger backoff timer will expire before the power-on message sent by the node with the smaller backoff timer is received.

5.2.3 Percentage of Coverage and Total Amount of Remaining Power

Zhang and Hou measure how coverage and the total remaining power changes over time. These are important metrics, as they show how long the network can stay alive and operational, and can be measured in Real-Time Maude simulations using analysis messages. The number of nodes in these simulations are reduced to 300 in [37], but still placed in a $50m \times 50m$ sensing area. Because of the long execution time of a large amount of nodes for several rounds, we scale down these parameters by $1:4$ to 75 nodes in a $25m \times 25m$ area. The reason why the authors in [37] reduce the number of nodes to 300 nodes is not known to us, but it could also be to reduce the execution time.

```
Maude> (tfrew {genInitConf(313, 75)
               dly(coveragePercentage(nil),roundTime - 1)
               dly(totalRemainingPower(nil),roundTime - 1)}
           in time <= roundTime * 50 .)

Result ClockedSystem :
  {dly(coveragePercentage(100 ++ 100 ++ 100 ++ 100 ++ 100 ++ 100 ++ 100 ++ 100 ++ 100
                          ++ 100 ++ 100 ++ 100 ++ 100 ++ 100 ++ 100 ++ 92 ++ 100 ++ 43
                          ++ 97 ++ 100 ++ 95 ++ 96 ++ 94 ++ 100 ++ 100 ++ 99 ++ 93 ++ 96
                          ++ 83 ++ 94 ++ 87 ++ 90 ++ 86 ++ 49 ++ 85 ++ 85 ++ 73 ++ 77 ++ 63
                          ++ 67 ++ 60 ++ 63 ++ 51 ++ 0 ++ 0 ++ 0 ++ 0 ++ 0 ++ 0 ++ 0 ++ 0),
       1000000)
   dly(totalRemainingPower(146337556845 ++ 140676548774 ++ 135021200518
                           ++  ...  ++ 3060430828 ++ 1497178694 ++ 365658984
                           ++ 0 ++ 0 ++ 0 ++ 0 ++ 0 ++ 0 ++ 0), 1000000)
  ... } in time 49999999
```

The result messages show that the nodes can provide 100% coverage for 25 rounds. However, the result shows a decrease of percentage of coverage in, e.g., rounds 16 and 18. The reason is probably that one or more nodes died in the middle of those rounds (the analysis messages compute their metrics at the *end* of each round). This causes the percentage of coverage to be temporarily decreased until the start of the next round, when new active nodes are selected. The last node in the network dies in round 44.

Simulations with different seeds showed 100% coverage for 21 to 28 rounds, and the last node dies after 39 to 43 rounds. In [37] the nodes provide 100% coverage for about 40 rounds, and the last node dies close to round 90. The results of the number of active nodes in Section 5.2.2 explains some of this discrepancy, since the more nodes that are active each round, the more power is consumed in the network.

5.3 Further Analysis of the OGDC Algorithm in Real-Time Maude

We give some examples of how we can further formally analyze the OGDC algorithm by examining all possible behaviors from a given initial state *relative to the treatment of probabilistic behaviors*, by using Real-Time Maude's time-bounded and untimed search and temporal logic model checking capabilities. Due to the large states involved, we restrict such analyses to systems with 5 to 6 nodes placed within a $15m \times 15m$ sensing area, which gives a fair chance of covering the sensing area and of getting sufficient overlap of the coverage areas so that some nodes can be switched off.

5.3.1 Reaching the Steady State Phase

The main task of a wireless sensor network is to perform its sensing task, which is performed in the steady state phase. This phase should, therefore, be reached at an early stage in each round of the OGDC algorithm so that most of the network's lifetime is used for sensing. We use the following `find latest` command to find the latest possible time the network enters the steady state phase (`such that steadyStatePhase(...)`), and thereby also find out whether this phase is always reached within the end of the round. The initial state below generates 5 nodes where two nodes volunteer to be starting node at the beginning of the round.

```
Maude> (find latest {genInitConf(1, 5)} =>* {C:Configuration}
                          such that steadyStatePhase(C:Configuration)
        in time <= roundTime .)

Result: { ... } in time 815
```

That is, the system will reach the steady state phase in *at most* 815 ms. Experimenting with many other initial seeds, the longest time we found was 1647 ms. The longest time for 6 nodes and initial seed 75 was 506 ms. One round of the OGDC algorithm is 1000 *seconds*, which means that the network spends most of its lifetime performing its sensing task. A similar search was done with the `find earliest` command. For initial seed 1, the steady state phase could be reached in 66 ms.

Another interesting property to investigate is whether or not the network stays in the steady state phase for the whole round, once this phase has been reached. We use Real-Time Maude's temporal logic model checker, and define an atomic proposition `steady-state-phase` to hold when the network is in steady state phase:

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

The following temporal logic formula checks whether all states following a state which is in the steady state phase are also in this phase[22].

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

Result Bool : true
```

5.3.2 Coverage in the Steady State Phase

The following time-bounded search command checks whether the entire sensing area is covered when the system is in steady state phase in the first round (when all the nodes have sufficient power to last until the end of the round):

[22] The notation A `=>` B is an abbreviation in (Real-Time) Maude for `[](`A `->` B`)`

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

The constant `sensingArea` is defined to be the sensing area bitmap. The function `updateArea` updates the bitmap by changing bits that are covered by the *active* nodes in the configuration `C` to `t`. The function `coverageAreaCovered` traverses the bitmap and returns `true` if, and only if, all the bits in the bitmap are set to `t` (Again, the formal definition of these functions can be found in [34] or [32]). The search command returned '`No solution`.'

5.3.3 A Node's Status and Coverage Area

To have a *minimal* set of active nodes, a node should be inactive when its coverage area is covered by other active nodes. The following search command searches for a state in which some node `O` is active (`status` is `on`) even though its coverage area is covered by other active nodes. The function `coveredBy` checks whether a node's coverage area is covered by traversing the configuration and checking whether its surrounding active nodes cover the node's coverage area.

```
Maude> (tsearch [1]
          {genInitConf(1,5)} =>*
              {< O:Oid : WSNode | status : on, bitmap : BM:BitMap, ATTS:AttributeSet >
               C:Configuration}
                  such that BM:Bitmap coveredBy C:Configuration
          in time <= roundTime .)

Solution 1
O:Oid <- -186 . 647 ;  ...
```

The result of the search is a state where the node at location `-186 . 647` is active even though the rest of the active nodes cover its entire coverage area. The reason is that this node became active *before* some of its neighbors did so. Nevertheless, the node's coverage area *is* covered and this node could be switched off without any loss of coverage. Therefore, there exist behaviors in the system where more nodes than necessary are active.

5.4 Summary of Our Analysis

We have, as suggested in an earlier version of [37], specified coverage areas as "bitmaps," and have emphasized ease and elegance over computational efficiency when defining bitmaps and functions on bitmaps. Although the coverage area of one node consists of 400 "bits," we could perform Monte Carlo simulation with 400 nodes in one round in less than an hour. We could also simulate 50 rounds of a system with 75 nodes in half an hour.[23] We exploited the ease by which messages with delays can be expressed in Real-Time Maude to define *analysis messages* to store "snapshots" of the system during its simulation. In this way, we could measure *all* performance metrics measured in the ns-2 simulations reported in [37]. The results of our simulations corresponded fairly well with the results of the ns-2 simulations, although we had in general more active nodes, and, consequently, better coverage and shorter network lifetime. 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 the OGDC algorithm would not break many ties when the transmission delay of a broadcast was taken into account.

The large bitmaps made exhaustive exploration of the reachable state space and temporal logic model checking infeasible for more than six nodes. However, analyzing networks with three nodes has been sufficient to find subtle bugs in other advanced Maude and Real-Time Maude applications [12, 25]. Our analysis did not find any design errors in the OGDC algorithm. It should again be emphasized that search and model checking only cover a fraction of the possible behaviors, since we have simulated the probabilistic behavior with pseudo-random numbers.

[23] We do not know how long the simulations in [37] took.

6 Concluding Remarks

In this paper we have shown how the challenging OGDC algorithm for wireless sensor networks was formally specified, simulated, and analyzed using Real-Time Maude. The Real-Time Maude specification captures the behavior of the algorithm at a high level of abstraction, and this specification—being precise, intuitive, and operational—could make a good starting point for an implementation of the OGDC algorithm on sensor networks.

Our specification was particularly suitable for *simulation* purposes. We defined *analysis messages* and could then perform the same simulations, and extract the same *performance metrics* from the simulations, as the algorithm developers did using the network simulation tool ns-2. Furthermore, 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. We have compared our simulation results with the results from the ns-2 simulations. In general, our simulations showed almost 50% worse performance of the OGDC algorithm than the ns-2 simulations.[24] The reason for this discrepancy is not clear, but we have pointed at some possible explanations.

Our work should continue in different directions. First, we focus on simplicity and elegance when modeling coverage areas and defining functions on such areas. It is not surprising that there is a price to pay in terms of longer execution times when we have hundreds of nodes, each of which contains a bitmap with 400 "bits." Therefore, much more efficient representations of coverage areas should be developed. This would also 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 probabilsitic properties, and that traditional reachability and model checking analyses are *incomplete*, since not all the behaviors in the OGDC algorithms are behaviors in our specification. We should therefore combine Real-Time Maude with methods and tools for probabilistic systems, such as the PMaude tool [1], and should develop new methods to fruitfully analyze probabilistic real-time specifications.

Finally, we should investigate the reason behind the different results of the Real-Time Maude simulations and the ns-2 simulations, so that we can be confident that Real-Time Maude can be useful and trustworthy for evaluating the performance of wireless sensor network algorithms. It would also be interesting to study the other wireless sensor network algorithms whose performance was compared to that of OGDC in [37], to see whether or not the reported differences in performance are mirrored in their Real-Time Maude analyses.

Acknowledgments.

We are grateful to Jennifer Hou for suggesting the OGDC algorithm as a challenging modeling task, and for discussions on sensor networks, and to José Meseguer for discussions on modeling communication in sensor networks. Part of the work reported here was performed while the second author was visiting the University of Illinois at Urbana-Champaign. Support for those visits by ONR Grant N00014-02-1-0715, by NSF Grant CCR-0234524, and by The Norwegian Research Council is gratefully acknowledged.

References

[1] G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *Proc. 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, 2005.

[2] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

[3] R. Barr, Z. Haas, and R. van Renesse. JiST: An efficient approach to simulation using virtual machines. *Software Practice and Experience*, 2004.

[4] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Proc. Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185

[24] However, we got better coverage.

of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004. See also UPPAAL home page at `http://www.uppaal.com`.

[5] M. Bozga, Susanne Graf, I. Ober, I. Ober, and J. Sifakis. Tools and applications II: The IF toolset. In M. Bernardo and F. Corradini, editors, *Proc. Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185, pages 237–267. Springer, 2004.

[6] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.

[7] N. Bulusu, J. Heidemann, D. Estrin, and T. Tran. Self-configuring localization systems: Design and experimental evaluation. Technical Report 8, Center for Embedded Networked Computing, University of California at Los Angeles,, September 2002. To appear in ACM TOCS.

[8] N. Bulusu, J. Heidemann, and D. Estrin. GPS-less low cost outdoor localization for very small devices. *IEEE Personal Communications Magazine*, 7(5):28–34, October 2000.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[10] M. Clavel, F. Dúran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1.1)*, April 2005. `http://maude.cs.uiuc.edu`.

[11] CMU monarch extensions to ns. `http://www.monarch.cs.cmu.edu/`.

[12] G. Denker, J. J. García-Luna-Aceves, J. Meseguer, P. C. Ölveczky, Y. Raju, B. Smith, and C. Talcott. Specification and analysis of a reliable broadcasting protocol in maude. In B. Hajek and R. S. Sreenivas, editors, *37th Annual Allerton Conference on Communication, Control, and Computation*. University of Illinois, 1999.

[13] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Fourth International Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[14] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, 2002.

[15] J. Elson and K. Römer. Wireless sensor networks: a new regime for time synchronization. *SIGCOMM Comput. Commun. Rev.*, 33(1):149–154, 2003.

[16] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[17] A. Hu and S.D. Servetto. Asymptotically optimal time synchronization in dense sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 1–10, New York, NY, USA, 2003. ACM Press.

[18] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, second edition, 1981.

[19] E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo, 2004.

[20] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[21] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[22] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[23] ns-2 network simulator. `http://www.isi.edu/nsnam/ns`.

[24] P. C. Ölveczky, M. Keaton, J. Meseguer, C. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2001.

[25] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004. Available at `http://www.ifi.uio.no/RealTimeMaude`.

[26] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[27] P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In T. Margaria and M. Wermelinger, editors, *Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *Lecture Notes in Computer Science*, pages 354–358. Springer, 2004.

[28] P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. In N. Martí-Oliet, editor, *Proc. Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 285–314. Elsevier, 2005.

[29] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of real-time maude. *Higher-Order and Symbolic Computation*, 2005. To appear.

[30] P. C. Ölveczky. *Real-Time Maude 2.1 Manual*, 2004. `http://www.ifi.uio.no/RealTimeMaude/`.

[31] P. C. Ölveczky. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude (extended version). `http://www.ifi.uio.no/RealTimeMaude/CASH`.

[32] Real-Time Maude home-page. `http://www.ifi.uio.no/RealTimeMaude/`.

[33] A. Savvides, C. Han, and M.B. Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Mobile Computing and Networking*, pages 166–179, 2001.

[34] S. Thorvaldsen. Modeling and analysis of the OGDC wireless sensor network algorithm in real-time maude. Master's thesis, University of Oslo, 2005.

[35] S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.

[36] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

[37] H. Zhang and J.C. Hou. Maintaining sensing coverage and connectivity in large sensor networks. *The Wireless Ad Hoc and Sensor Networks: An International Journal*, 2005.

A The Complete Real-Time Maude Specification of OGDC

The following presents the entire executable Real-Time Maude specification of the OGDC algorithm.

```
****( TUNABLE PARAMETERS )****

(omod OGDC-PARAMETERS is
    protecting CONVERSION .
    protecting NAT-TIME-DOMAIN-WITH-INF .

    ---( Constants )---
    --- Number of nodes in network
    op n : -> Nat .

    --- Size of sensor area in meters
    op sensingAreaSize : -> Nat .

    --- Round time in ms
    op roundTime : -> Nat .    eq roundTime = 1000000 .

    --- If a node does not volunteer it sets it's timer to this value
    op nonVolunteerTimer : -> Time .    eq nonVolunteerTimer = 1000 .
    --- Upper bound for the volunteer timer
    op volunteerTimeBound : -> Time .   eq volunteerTimeBound = 10 .

    --- Timer is set to this value when a node receives a power-on
    --- message and all crossings within range are covered and
    --- there are no starting neighbors
    op tc : -> Time .    eq tc = 200 .

    --- Sensing range in centimeters
    op sensingRange : -> Nat .       eq sensingRange = 1000 .

    --- Transmission range in centimeters
    op transmissionRange : -> Nat .  eq transmissionRange = 2 * sensingRange .

    --- Transmission time in ms (originally 6.8)
    op transmissionDelay : -> Nat .  eq transmissionDelay = 7 .

    --- Backoff scale
    op c : -> Rat .      eq c = 10 / (sensingRange * sensingRange) .

    ---( Power related constants )---
    --- Power ratio = transmit:idle:sleep = 5:1:0.0025

    --- One power unit (pu) is the amount of power needed
    --- to stay idle for one millisecond
    op powerUnit : -> Nat .          eq powerUnit = 400 .

    --- Startup power, lifetime
    op lifetime : -> Nat .           eq lifetime = 5000000 * powerUnit .

    --- Power threshold - minimum power required to volunteer---
    op powerThreshold : -> Nat .     eq powerThreshold = 900000 * powerUnit .

    --- Power required to transmit one power-on message
    op transPower : -> Nat .
    eq transPower = powerUnit * 5 * transmissionDelay .

    --- Power required to sleep one millisecond
    op sleepPower : -> Nat .         eq sleepPower = powerUnit * (25 / 10000) .

    --- Power required to stay idle for one millisecond
    op idlePower : -> Nat .          eq idlePower = powerUnit .
endom)



****( BASIC SORTS AND FUNCTIONS )****

(omod OGDC-SORTS is
    protecting OGDC-PARAMETERS .
    including BOOL .
```

```
    sorts LocationSet Location Status VolunteeredStatus NeighborSet Neighbor .

    subsort Location < Oid LocationSet .
    subsort Neighbor < NeighborSet .
    subsort Bool < VolunteeredStatus .

    --- A location is represented by its coordinates
    op _._ : Rat Rat -> Location [ctor] .

    --- Definition of a set of locations and some basic function on this set
    op none    : -> LocationSet [ctor] .
    op _;_     : LocationSet LocationSet -> LocationSet [ctor assoc comm id: none] .
    op _minus_ : LocationSet LocationSet -> LocationSet .
    op remove  : LocationSet Oid -> LocationSet .

    --- A node can have one of three statuses
    op on        : -> Status [ctor format ( g o )] .
    op off       : -> Status [ctor format ( r o )] .
    op undecided : -> Status [ctor] .

    --- A node can have one of three volunteered statuses; undecided, true, or false
    op undecided : -> VolunteeredStatus [ctor] .

    --- A neighbor is represented by it's coordinate and whether it's a starting node
    op _starting_ : Location Bool -> Neighbor .
    op none       : -> NeighborSet [ctor] .
    op __         : NeighborSet NeighborSet -> NeighborSet [ctor assoc comm id: none] .

    --- RandomNGen object name
    op Random : -> Oid .

    --- Basic functions for computing and checking distances
    op isInSensingArea                : Location -> Bool .
    op vectorLengthSq                 : Location Location -> Rat .
    op _withinSensingRangeOf_         : Location Location -> Bool .
    op _withinTwiceTheSensingRangeOf_ : Location Location -> Bool .
    op _withinTransmissionRangeOf_    : Location Location -> Bool .


    vars L L'      : Location .
    vars LS LS'    : LocationSet .
    vars X X' Y Y' : Rat .
    var SASIZE     : Nat .


    ceq isInSensingArea(X . Y) = (abs(X) <= SASIZE) and (abs(Y) <= SASIZE)
     if SASIZE := sensingAreaSize / 2 .

    eq vectorLengthSq(X . Y, X' . Y') = ((X - X') * (X - X')) + ((Y - Y') * (Y - Y')) .

    eq L withinSensingRangeOf L' = vectorLengthSq(L, L') <= (sensingRange * sensingRange) .

    eq L withinTwiceTheSensingRangeOf L' = vectorLengthSq(L, L') <= (4 * sensingRange * sensingRange) .

    eq L withinTransmissionRangeOf L' = vectorLengthSq(L, L') <= (transmissionRange * transmissionRange) .

    eq LS minus none = LS .
    eq none minus LS = none .
    eq LS minus (L ; LS') = remove(LS, L) minus LS' .

    eq remove(none, L) = none .
    eq remove((LS ; L), L') = (if (L == L') then remove(LS, L') else remove(LS, L') ; L fi) .
endom)



****( BITMAP DEFINITION ****

(omod OGDC-BITMAP is
    protecting OGDC-SORTS .
```

```
    sorts Bitmap BitList Bit .
    subsort Bit < BitList .

    --- A cell in the bitmap is either true (t) false (f) or not in use (')
    op t : -> Bit [ctor format ( g o )] .
    op f : -> Bit [ctor format ( r o )] .
    op ' : -> Bit [ctor format ( y o )] .

    --- Collect bit's into a list
    op nil : -> BitList [ctor] .
    op __  : BitList BitList -> BitList [ctor assoc id: nil format ( o s o )] .

    --- Incapsulate one row in the bitmap with | | starting on a
    --- separate line to give more intuitive formatting
    op |_| : BitList -> Bitmap [ctor format ( ni o o o )] .

    --- Collect rows into a bitmap
    op nil : -> Bitmap [ctor] .
    op __  : Bitmap Bitmap -> Bitmap [ctor assoc id: nil format ( o s o )] .

    --- Define the distance between each bit
    op gridInc : -> Nat .  eq gridInc = 100 .

    --- Determine how many rows and columns the bitmap will have
    op gridCounter  : -> Nat .
    eq gridCounter = ceiling((2 * sensingRange) / gridInc) +
                     if (gridInc divides (2 * sensingRange))
                     then 1 else 0
                     fi .

    --- Initialize bitmap
    op initBitmap       : Location -> Bitmap .
    op makeColumnBitmap : Location Location Nat Nat -> Bitmap .
    op makeRowBitmap    : Location Location Nat Nat -> BitList .

    --- Update bitmap
    op updateBitmap  : Location Bitmap Location -> Bitmap .
    op updateBitmap  : Bitmap Location Location Nat -> Bitmap .
    op updateBitList : BitList Location Location Nat -> BitList .

    --- Check if sensor area is completely covered
    op coverageAreaCovered : Bitmap -> Bool .


    vars L L'      : Location .
    vars BM        : Bitmap .
    var BITL       : BitList .
    vars X X' Y Y' : Rat .
    var BIT        : Bit .
    vars I N       : Nat .


    --- Start making the bitmap from the top left corner
    eq initBitmap(X . Y) =
        makeColumnBitmap(X . Y, X - sensingRange . Y + sensingRange, gridInc, gridCounter) .

    --- Make bitmap centered in L, with N rows and columns and increment I
    eq makeColumnBitmap(L, X . Y, I, N) =
        if N > 0
        then | makeRowBitmap(L, X . Y, I, gridCounter) |
     makeColumnBitmap(L, X . Y - I, I, N - 1)
else nil
fi .

    eq makeRowBitmap(X' . Y', X . Y, I, N) =
        if N > 0
        then (if ((X' . Y') withinSensingRangeOf (X . Y))
                  and isInSensingArea(X . Y)
      then f else '  fi)
```

```
     makeRowBitmap(X' . Y', X + I . Y, I, N - 1)
        else nil
fi .

    --- Update the bitmap from the top left corner
    eq updateBitmap(X . Y, BM, L) =
           updateBitmap(BM, L, X - sensingRange . Y + sensingRange, gridInc) .

    eq updateBitmap(nil, L', L, I) = nil .
    eq updateBitmap(| BITL | BM, L', X . Y, I) =
         | updateBitList(BITL, L', X . Y, I) |
 updateBitmap(BM, L', X . Y - I, I) .

    --- Update one row of the bitmap
    eq updateBitList(nil, L', L, I) = nil .
    eq updateBitList(BIT BITL, L', X . Y, I) =
        if (BIT =/= f)
then BIT
        else if L' withinSensingRangeOf (X . Y) then t else f fi
        fi
updateBitList(BITL, L', X + I . Y, I) .

    --- Check if sensing area is completely covered after updated
    --- with the latest neighbor's position
    eq coverageAreaCovered(nil) = true .
    eq coverageAreaCovered(| nil | BM) = coverageAreaCovered(BM) .
    eq coverageAreaCovered(| BIT BITL | BM) =
              (BIT =/= f) and  coverageAreaCovered(| BITL | BM) .
endom)


****( COMPUTATION OF CROSSINGS ****

(omod OGDC-COVERAGE-OPERATIONS is
    protecting OGDC-SORTS .

    sorts Crossing CrossingSet .
    subsort Crossing < CrossingSet .

    --- A crossing between two sensing areas is represented by the location of the two nodes
    --- that create the crossing and the coordinates of the crossing
    op _x_in_ : Location Location Location -> Crossing [ctor] .
    op none   : -> CrossingSet [ctor] .
    op __     : CrossingSet CrossingSet -> CrossingSet [ctor assoc comm id: none]  .

    --- Find all crossings created by a node at a given location and a list of neighbors
    op newCrossings  : Location Location NeighborSet -> CrossingSet .
    op findCrossings : Location NeighborSet -> CrossingSet .

    --- Update a list of uncovered crossings
    op updateUncoveredCrossings : Location Location NeighborSet CrossingSet -> CrossingSet .

    --- Extract uncovered crossings
    op listUncoveredCrossings : LocationSet CrossingSet -> CrossingSet .
    op crossingIsUncovered    : LocationSet Crossing -> Bool .

    --- Extract crossings that are within range of a given location
    op extractValidCrossings : Location CrossingSet -> CrossingSet .

    --- Determine whether a location creates the closest crossing to another location
    op _createsClosestCrossing__    : Location Location CrossingSet -> Bool .
    op closestCrossing              : Location CrossingSet -> Crossing .

    --- Find closest starting neighbor
    op findClosestStartingNeighbor : Location NeighborSet -> Location .

    --- Determine wether there exists a starting neighbor in a list of neighbors
    op existsStartingNeighbor      : NeighborSet -> Bool .

    --- Extract all starting neighbors
```

```
  op listStartingNeighbors       : NeighborSet -> LocationSet .

  --- Find the closest neighbor to a location
  op findClosestNeighbor         : Location NeighborSet -> Location .
  op compareNeighbors            : Location LocationSet -> Location .

  --- Compute the crossings between two nodes
  op findPairCrossings  : Location Location -> CrossingSet .
  op calcLinearCrossing : Location Location Bool -> Location .

  --- Auxiliary operators to calculate a crossing point
  op  sqrt              : Rat -> Rat .
  ops delta u v a b c   : Location Location -> Rat .
  ops y x               : Location Location Bool -> Rat .

  --- Other auxiliary functions
  op extractLocation         : NeighborSet -> LocationSet .
  op locationCreatesCrossing : Location Crossing -> Bool .
  op extractCrossing         : Crossing -> Location .


  var B               : Bool .
  vars LS             : LocationSet .
  vars L L' L'' L'''
       L1' L1'' L1''' : Location .
  vars X X' Y Y' R    : Rat .
  vars C              : Crossing .
  vars CS             : CrossingSet .
  var NBS             : NeighborSet .


  eq sqrt(R) = rat(sqrt(float(R))) .

  eq L' createsClosestCrossing L CS = locationCreatesCrossing(L',
 closestCrossing(L, CS)) .

  --- Find closest crossing
  eq closestCrossing(L, none) = none .
  eq closestCrossing(L, C) = C .
  eq closestCrossing(L, (L' x L'' in L''') (L1' x L1'' in L1''') CS) =
      if vectorLengthSq(L, L''') < vectorLengthSq(L, L1''')
      then closestCrossing(L, (L' x L'' in L''') CS)
      else closestCrossing(L, (L1' x L1'' in L1''') CS)
      fi .

  --- Update the list of uncovered crossings
  eq updateUncoveredCrossings(L, L', NBS, CS) =
          listUncoveredCrossings(L', CS)
          listUncoveredCrossings(extractLocation(NBS), newCrossings(L, L', NBS)) .

  eq newCrossings(L, L', NBS) = extractValidCrossings(L, findCrossings(L', NBS)) .

  eq findCrossings(L, none) = none .
  eq findCrossings(L, (L' starting B) NBS) =
       (if L withinTwiceTheSensingRangeOf L'
 then findPairCrossings(L, L')
 else none
fi)
       findCrossings(L, NBS) .

  --- Only keep crossings that are uncovered by a third node
  eq listUncoveredCrossings(LS, none) = none .
  eq listUncoveredCrossings(LS, (L x L' in L'') CS) =
      if crossingIsUncovered(LS minus (L ; L'), (L x L' in L''))
      then (L x L' in L'')
      else none
      fi
      listUncoveredCrossings(LS, CS) .

  --- Determine if a crossing is within range of the nodes in LS
```

```
    eq crossingIsUncovered(none, C) = true .
    eq crossingIsUncovered(L ; LS, (L' x L'' in L''')) =
        (not L withinSensingRangeOf L''')
        and crossingIsUncovered(LS, (L' x L'' in L''')) .

    ---Only keep the crossings within sensor range of L
    eq extractValidCrossings(L, none) = none .
    eq extractValidCrossings(L, (L' x L'' in L''') CS) =
        if L withinSensingRangeOf L'''
then (L' x L'' in L''')
else none
fi
extractValidCrossings(L, CS) .

    --- Find the crossings between two nodes
    eq findPairCrossings(X . Y, X' . Y') =
         if X =/= X'

         then (X . Y x X' . Y' in x(X . Y, X' . Y', true) . y(X . Y, X' . Y', true))
              (X . Y x X' . Y' in x(X . Y, X' . Y', false) . y(X . Y, X' . Y', false))

         else (X . Y x X' . Y' in calcLinearCrossing(X . Y, X' . Y', true))
              (X . Y x X' . Y' in calcLinearCrossing(X . Y, X' . Y', false))
         fi .

    --- Special case where the one node is directly above the other
    eq calcLinearCrossing(X . Y, X' . Y', true)  = ((2 * X + sqrt(
                (((4 * sensingRange * sensingRange) - (Y' * Y')) + (2 * Y * Y')) -
                  (Y * Y))) / 2  . (Y + Y') / 2) .
    eq calcLinearCrossing(X . Y, X' . Y', false) = ((2 * X - sqrt(
                (((4 * sensingRange * sensingRange) - (Y' * Y')) + (2 * Y * Y')) -
                   (Y * Y))) / 2  . (Y + Y') / 2) .

    --- Compute the x and y coordinate
    eq y(L, L', true)  = ceiling((- b(L, L') + delta(L, L')) / (2 * a(L, L'))) .
    eq y(L, L', false) = ceiling((- b(L, L') - delta(L, L')) / (2 * a(L, L'))) .

    eq x(L, L', B) = ceiling((- u(L, L') * y(L, L', B)) - v(L, L')) .

    eq delta(L, L') = sqrt((b(L, L') * b(L, L')) - (4 * a(L, L') * c(L, L'))) .

    eq u(X . Y, X' . Y') = (Y - Y') / (X - X') .

    eq v(X . Y, X' . Y') = (((X' * X') + (Y' * Y')) - ((X * X) + Y * Y))
                                      / (2 * (X - X')) .

    --- ay2 + by + c = 0
    eq a(L, L') = 1 + (u(L, L') * u(L, L')) .

    eq b(L, X' . Y') = 2 * u(L, X' . Y') * (X' + v(L, X' . Y')) - (2 * Y') .

    eq c(L, X' . Y') =
         ((Y' * Y') + ((v(L, X' . Y') + X') * (v(L, X' . Y') + X')))
                       - (sensingRange * sensingRange) .

    --- Determine if a starting neighbor exists
    eq existsStartingNeighbor(none) = false .
    eq existsStartingNeighbor((X . Y starting B) NBS) =
             B or  existsStartingNeighbor(NBS) .

    --- Find the closest starting neighbor to L in NBS
    eq findClosestStartingNeighbor(L, NBS) =
             compareNeighbors(L, listStartingNeighbors(NBS)) .

    --- List all starting neighbors
    eq listStartingNeighbors(none) = none .
    eq listStartingNeighbors((X . Y starting B) NBS) =
             if B then X . Y else none fi
     ; listStartingNeighbors(NBS) .
```

```
    --- Find the closest neighbor to L in NBS
    eq findClosestNeighbor(L, NBS) = compareNeighbors(L, extractLocation(NBS)) .

    --- Find the closest neighbor to L in LS
    eq compareNeighbors(L, L') = L' .
    eq compareNeighbors(L, L' ; L'' ; LS) =
             if vectorLengthSq(L, L') < vectorLengthSq(L, L'')
     then compareNeighbors(L, L' ; LS)
     else compareNeighbors(L, L'' ; LS)
     fi .

    eq extractLocation(none) = none .
    eq extractLocation((L starting B) NBS) = L ; extractLocation(NBS) .

    eq locationCreatesCrossing(L, L' x L'' in L''') = L == L' or L == L'' .

    eq extractCrossing(L x L' in L'') = L'' .
endom)



****( RANDOM NUMBER GENERATOR )****

(omod RANDOM is
  including NAT .

  class RandomNGen | seed : Nat .

  op random : Nat -> Nat .      --- random(x) generates the next random number

  vars N N' : Nat .

  eq random(N) = ((104 * N) + 7921) rem 10609 .
  --- Obeys Knuths criteria for a "good" random function

  --- The seed may be modified by applying the random function many times:
  op repeatRandom : Nat Nat -> Nat .     --- repeatRandom(seed, noOfReps)

  eq repeatRandom(N, s N') = repeatRandom(random(N), N') .
  eq repeatRandom(N, 0) = N .
endom)



****( BACKOFF TIMER COMPUTATIONS )****

(omod OGDC-TC is
    protecting OGDC-COVERAGE-OPERATIONS .
    protecting RANDOM .

    op setTa : Location Nat CrossingSet -> Time .
    --- Angle between optimal location and location of receiver node
    op dAlphaTa               : Location Crossing -> Rat .
    op dAlphaTaSingleCrossing : Location Crossing -> Rat .
    ---Distance from receiver to closest crossing point
    op dTa : Location Crossing -> Rat .

    op setTb    : Location Location Nat NeighborSet Nat -> Time .
    --- Angle between desired node position and receiver node
    op dAlphaTb : Location Location Nat -> Rat .
    ---Distance from receiver to sender---
    op dTb      : Location Location -> Rat .


    op acos : Rat -> Rat .
    eq acos(R:Rat) = rat(acos(float(R:Rat))) .

    op pi   :     -> Rat .
    eq pi = rat(pi) .
```

```
    op negY : Location -> Bool .

    --- Calculate the angle the vector created by the two locations and the x-axis
    op angle     : Location Location -> Rat .
    --- Normalize a vector
    op normalize : Location -> Location .
    --- Calculate the scalar product of two vectors
    op dotProd   : Location Location -> Rat .

    --- Random term [0 , 1>
    op randomU : Nat -> Rat .
    eq randomU(N:Nat) = (random(N:Nat) rem 100) / 100 .


    vars R ANGLE X X' Y Y' : Rat .
    vars N D               : Nat .
    vars C CLOSESTC        : Crossing .
    var CS                 : CrossingSet .
    var NBS                : NeighborSet .
    vars L L' VECTOR       : Location .


    ceq setTa(L, N, CS) = ceiling(transmissionDelay * (c *
              (((sensingRange - dTa(L, CLOSESTC)) * (sensingRange - dTa(L, CLOSESTC)))
              + (dTa(L, CLOSESTC) * dTa(L, CLOSESTC) * dAlphaTa(L, CLOSESTC) * dAlphaTa(L, CLOSESTC)))
        + randomU(N)))
     if CLOSESTC := closestCrossing(L, CS) .

    --- Distance between receiver node and the closest uncovered crossing point
    eq dTa(L, C) = sqrt(vectorLengthSq(L, extractCrossing(C))) .

    ---( Angle between the optimal node location and the location of
    --- the receiving node )---
    eq dAlphaTa(L, C) = dAlphaTaSingleCrossing(L, C) .

    ceq dAlphaTaSingleCrossing(L, X . Y x X' . Y' in L') =
(if ANGLE > pi
 then 2 * pi - ANGLE
 else ANGLE
fi)
      if ANGLE := abs(angle(L', L) - angle(((X' + X) / 2 . (Y' + Y) / 2), L')) .


    eq setTb(L, L', D, NBS, N) = ceiling(transmissionDelay * (c *
               (((sqrt(3) * sensingRange - dTb(L, L')) * (sqrt(3) * sensingRange - dTb(L, L')))
               + (dTb(L, L') * dTb(L, L') * dAlphaTb(L, L', D) * dAlphaTb(L, L', D)))
       + randomU(N))) .

    --- Distance from sender to receiver
    eq dTb(L, L') = sqrt(vectorLengthSq(L, L')) .

    --- Angle between the desired direction in which the node should be
    --- located and the direction from the sender to the receiver
    ceq dAlphaTb(L, L', D) = (if ANGLE > pi then 2 * pi - ANGLE else ANGLE fi)
      if ANGLE := abs(angle(L', L) - ((D / 180) * pi)) .


    --- Angle between the vector from (X . Y) to (X' . Y') and the x-axis
    ceq angle(X . Y, X' . Y') =
     (if negY(VECTOR)
      then 2 * pi - acos(dotProd(VECTOR, 1 . 0))
      else acos(dotProd(VECTOR, 1 . 0))
     fi)
      if VECTOR := normalize((X' - X) . (Y' - Y)) .

    eq normalize(X . Y) = ((X / (sqrt((X * X) + (Y * Y))))
 . (Y / (sqrt((X * X) + (Y * Y))))) .

    eq dotProd(X . Y, X' . Y') = (X * X') + (Y * Y') .
```

```
    eq negY(X . Y) = Y < 0 .
endom)



****( NODE AND MESSGAE DEFINITIONS )****

(tomod OGDC-NODE-AND-MESSAGE-DEFINITIONS is
    protecting OGDC-BITMAP .
    protecting OGDC-TC .

    --- Messages
    msg broadcast'from_withDirection_     : Oid Int -> Msg .
    msg power-onMsgFrom_to_withDirection_ : Oid Oid Int -> Msg .
    op dly : Msg TimeInf -> Msg [ctor right id: 0] .

    --- Wireless Sensor Node
    class WSNode | backoffTimer : TimeInf,
   bitmap : Bitmap,
   hasVolunteered : VolunteeredStatus,
                    neighbors : NeighborSet,
   uncoveredCrossings : CrossingSet,
   remainingPower : Nat,
   roundTimer : TimeInf,
   status : Status,
                    volunteerProb : Rat .
endtom)



****( THE OGDC ALGORITHM )****

(tomod OGDC is
    protecting OGDC-NODE-AND-MESSAGE-DEFINITIONS .

    --- Message distribution
    op distributeMsg : Oid Nat Configuration -> Configuration [frozen (3)] .

    --- Auxiliary fuctions
    --- Double the probability for volunteering
    op doubleProb  : Rat -> Rat .
    --- [0, 1000>
    op randomProb  : Nat -> Nat .
    --- [0, td>
    op randomTimer : Nat -> Time .
    --- [0, 360>
    op randomDirection : Nat -> Nat .


    vars O O' : Oid .
    var D     : Int .
    vars M N  : Nat .
    var P     : NzNat .
    vars R    : Rat .
    var L     : Location .
    var BM    : Bitmap .
    var NB    : Neighbor .
    var NBS   : NeighborSet .
    var CS    : CrossingSet .
    vars T    : Time .
    var S     : Status .
    var C     : Configuration .
    var MSG   : Msg .


    --- Message broadcasting
    eq {< O : WSNode | > (broadcast from O withDirection D) C} =
       {< O : WSNode | > distributeMsg(O, D, C)} .

    --- Break down the broadcast message to single messages for nodes which
```

```
  --- are within transmission range
  eq distributeMsg(O, D, none) = none .

  eq distributeMsg(O, D, MSG C) = MSG distributeMsg(O, D, C) .

  eq distributeMsg(O, D, < Random : RandomNGen | > C) =
        < Random : RandomNGen | > distributeMsg(O, D, C) .

  eq distributeMsg(O, D, < O' : WSNode | > C) =
< O' : WSNode | > distributeMsg(O, D, C)
        (if O withinTransmissionRangeOf O'
 then dly((power-onMsgFrom O to O' withDirection D), transmissionDelay)
 else none
fi) .


  eq doubleProb(R) = if R >= 1000 / 2 then 1000 else 2 * R fi .

  eq randomProb(N) = random(N) rem 1000 .
  eq randomTimer(N) = random(N) rem volunteerTimeBound .
  eq randomDirection(N) = random(N) rem 360 .


  ---( Dynamic behavior in OGDC )---

  ---( A node volunteers with probability N )---
  rl [volunteer] :
      < O : WSNode | remainingPower : P, volunteerProb : R, hasVolunteered : undecided >
      < Random : RandomNGen | seed : M >
    =>
      (if (randomProb(M) < R) and (P > powerThreshold or R == 1000)
       then < O : WSNode | backoffTimer : randomTimer(random(M)), hasVolunteered : true >
       else < O : WSNode | backoffTimer : nonVolunteerTimer, hasVolunteered : false,
  volunteerProb : doubleProb(R) >
      fi)
      < Random : RandomNGen | seed : repeatRandom(M, 3) > .


  --- If the nonVolunteerTimer timer expires, volunteer again with the volunteer
  --- probability doubled
  rl [repeatVolunteering] :
     < O : WSNode | backoffTimer : 0, neighbors : none, volunteerProb : R,
                    remainingPower : P, hasVolunteered : false >
      < Random : RandomNGen | seed : M >
    =>
      (if (randomProb(M) < R) and (P > powerThreshold or R == 1000)
       then < O : WSNode | backoffTimer : randomTimer(random(M)), hasVolunteered : true >
       else < O : WSNode | backoffTimer : nonVolunteerTimer, volunteerProb : doubleProb(R) >
      fi)
      < Random : RandomNGen | seed : repeatRandom(M, 3) > .


  --- Power on and send out a message when the timer expires
  rl [startingNodePowerOn] :
     < O : WSNode | remainingPower : P, backoffTimer : 0, hasVolunteered : true >
     < Random : RandomNGen | seed : M >
    =>
     < O : WSNode | remainingPower : P monus transPower, backoffTimer : INF, status : on >
     < Random : RandomNGen | seed : random(M) >
     broadcast from O withDirection randomDirection(M) . --- become active  as starting node


  --- Power on and send out a message when the timer expires
  rl [nonStartingNodePowerOn] :
     < O : WSNode | remainingPower : P, backoffTimer : 0,
                    neighbors : NB NBS, hasVolunteered : false >
    =>
     < O : WSNode | remainingPower : P monus transPower,
                    backoffTimer : INF, status : on >
     broadcast from O withDirection -1 . --- become active as non-starting node
```

```
    --- Power off if remainingPower : 0
    ceq < O : WSNode | status : S, remainingPower : 0 > =
        < O : WSNode |  backoffTimer : INF, roundTimer : INF,
                        status : off, hasVolunteered : false >
     if S =/= off .


    --- Power off if O's neighbors completely cover its sensing area
    crl [recPowerOnMsgAndSwichOff] :
(power-onMsgFrom O' to O withDirection D)
< O : WSNode | status : undecided, neighbors : NBS, bitmap : BM,
               remainingPower : P >
      =>
       < O : WSNode | status : off, neighbors : NBS (O' starting (D >= 0)),
       bitmap : updateBitmap(O, BM, O'), backoffTimer : INF >
      if O withinTwiceTheSensingRangeOf O' /\
         coverageAreaCovered(updateBitmap(O, BM, O')) .


    --- Update timer to Ta if O' creates closest uncovered crossing
    crl [recPowerOnWithUncoveredCrossings] :
(power-onMsgFrom O' to O withDirection D)
< O : WSNode | remainingPower : P, status : undecided, backoffTimer : T,
       neighbors : NBS, uncoveredCrossings : CS, bitmap : BM >
        < Random : RandomNGen | seed : M >
      =>
        < O : WSNode | backoffTimer : (if O' createsClosestCrossing
                                                 O (updateUncoveredCrossings(O, O', NBS, CS))
                                       then setTa(O, M, updateUncoveredCrossings(O, O', NBS, CS))
                                       else T
                                       fi),
                       neighbors : NBS (O' starting (D >= 0)),
       uncoveredCrossings : updateUncoveredCrossings(O, O', NBS, CS),
                    bitmap : updateBitmap(O, BM, O') >
        < Random : RandomNGen | seed : random(M) >

      if O withinTwiceTheSensingRangeOf O' /\
         updateUncoveredCrossings(O, O', NBS, CS) =/= none /\
 not coverageAreaCovered(updateBitmap(O, BM, O')) .


    --- Update timer to Tb if all crossings are covered and
    --- O' is the closest starting neighbor
    crl [recPowerOnWithStartingNeighbors] :
(power-onMsgFrom O' to O withDirection D)
< O : WSNode | remainingPower : P, status : undecided, backoffTimer : T,
       neighbors : NBS, uncoveredCrossings : CS, bitmap : BM >
        < Random : RandomNGen | seed : M >
      =>
        < O : WSNode | backoffTimer : (if O' == findClosestStartingNeighbor(O,
            NBS (O' starting (D >= 0)))
                                       then setTb(O, O', D, NBS (O' starting (D >= 0)), M)
                                       else T
                                       fi),
                       neighbors : NBS (O' starting (D >= 0)),
       uncoveredCrossings : none,
                bitmap : updateBitmap(O, BM, O') >
        < Random : RandomNGen | seed : random(M) >

      if O withinTwiceTheSensingRangeOf O' /\
         existsStartingNeighbor(NBS (O' starting (D >= 0))) /\
 updateUncoveredCrossings(O, O', NBS, CS) == none /\
 not coverageAreaCovered(updateBitmap(O, BM, O')) .


    --- Update timer to TC if all crossings are covered and
    --- there's no starting neighbor and
    --- O' is the closest neighbor
```

```
    crl [recPowerOnWithNeighbors] :
(power-onMsgFrom O' to O withDirection D)
< O : WSNode | remainingPower : P, status : undecided, backoffTimer : T,
       neighbors : NBS, uncoveredCrossings : CS, bitmap : BM >
      =>
        < O : WSNode | backoffTimer : (if O' == findClosestNeighbor(O,
  NBS (O' starting (D >= 0)))
                                       then tc
                                       else T
                                       fi),
                       neighbors : NBS (O' starting (D >= 0)),
       uncoveredCrossings : none,
                   bitmap : updateBitmap(O, BM, O') >

      if O withinTwiceTheSensingRangeOf O' /\
         not existsStartingNeighbor(NBS (O' starting (D >= 0))) /\
 updateUncoveredCrossings(O, O', NBS, CS) == none /\
 not coverageAreaCovered(updateBitmap(O, BM, O')) .


    --- Discard/ignore power-on messages received after powered on
    crl [discard] :
(power-onMsgFrom O' to O withDirection D)
< O : WSNode | status : S >
      =>
        < O : WSNode | >
      if S =/= undecided or not O withinTwiceTheSensingRangeOf O' .


    --- Discard/ignore power-on messages if the node is dead
    eq (power-onMsgFrom O' to O withDirection D)
       < O : WSNode | status : off, remainingPower : 0 > =
< O : WSNode | > .


    --- Restart the node when the round is over
    rl [restart] :
       < O : WSNode | roundTimer : 0, remainingPower : P >
     =>
       < O : WSNode | status : undecided,
                      neighbors : none,
              uncoveredCrossings : none,
      bitmap : initBitmap(O),
      hasVolunteered : undecided,
      backoffTimer : INF,
      roundTimer : roundTime,
      volunteerProb : 1000 / n > .
endtom)


****************************************
********* REAL TIME BEHAVIOR *************
****************************************

(tomod OGDC-RTM is
    protecting OGDC .

    var O       : Oid .
    vars T T'   : Time .
    vars TI TI' : TimeInf .
    var P       : NzNat .
    var R       : Rat .
    var S       : Status .
    var V       : VolunteeredStatus .
    var M       : Msg .


    crl [tick] :
        {C:Configuration}
      =>
```

```
     {delta(C:Configuration, T)} in time T
      if T <= mte(C:Configuration) [nonexec] .


    ---( Delta )---
    op delta : Configuration Time -> Configuration [frozen (1)] .
    eq delta(none, T') = none .
    eq delta(NEC:NEConfiguration NEC':NEConfiguration, T') =
           delta(NEC:NEConfiguration, T') delta(NEC':NEConfiguration, T') .

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

    eq delta(< Random : RandomNGen | >, T') = < Random : RandomNGen | > .
    eq delta(dly(M, TI), T') = dly(M, TI monus T') .


    ---( Maximum Time Elapse )---
    op mte : Configuration -> TimeInf [frozen (1)] .
    eq mte(none) = INF .
    eq mte(NEC:NEConfiguration NEC':NEConfiguration) =
            min(mte(NEC:NEConfiguration), mte(NEC':NEConfiguration)) .


    eq mte(< O : WSNode | remainingPower : 0, status : S >) = if S == off then INF else 0 fi .

    eq mte(< O : WSNode | backoffTimer : TI, roundTimer : T, remainingPower : P,
                          hasVolunteered : true, status : S >) =
            min(TI, T, if S == on then ceiling(P / powerUnit) else P fi) .

    eq mte(< O : WSNode | backoffTimer : TI, roundTimer : T, remainingPower : P,
                          hasVolunteered : false, status : S >) =
            min(TI, T, if S == on then ceiling(P / powerUnit) else P fi) .

    eq mte(< O : WSNode | hasVolunteered : undecided >) = 0 .

    eq mte(< Random : RandomNGen | >) = INF .

    eq mte(dly(M, TI)) = TI .
endtom)


****************************************
********* ANALYSIS ***********************
****************************************

(tomod OGDC-ANALYSIS is
    protecting OGDC-RTM .
    including TIMED-MODEL-CHECKER .


    --- The size of the sensing area in cm (5000 denotes 50mx50m area)
    eq sensingAreaSize = 5000 .
    --- The number of nodes in the system
    eq n = 200 .


    vars M N N' P : Nat .
    var NL        : NatList .
    var D         : Int .
    var T         : Time .
    var BIT       : Bit .
```

```
var BITL      : BitList .
var BM        : Bitmap .
var LS        : LocationSet .
var O         : Oid .
var L         : Location .
var S         : Status .
var NB        : Neighbor .
var NBS       : NeighborSet .
var SYSTEM    : Configuration .
var MSG       : Msg .


---( Definition of analysis messages )---

--- List of natural numbers
sort NatList .
subsort Nat < NatList .

op nil : -> NatList .
op _++_ : NatList NatList -> NatList [ctor assoc id: nil] .

--- The analysis messages
msg activeNodes         : NatList -> Msg .
msg coveragePercentage  : NatList -> Msg .
msg totalRemainingPower : NatList -> Msg .


rl [computeActiveNodes] :
   {activeNodes(NL) SYSTEM}
 =>
   {dly(activeNodes(NL ++ numActiveNodes(SYSTEM)), roundTime) SYSTEM} .

rl [computeSensingCoverage] :
   {coveragePercentage(NL) SYSTEM}
 =>
   {dly(coveragePercentage(NL ++ coveragePercentage(SYSTEM)), roundTime) SYSTEM} .

rl [computeTotalPower] :
   {totalRemainingPower(NL) SYSTEM}
 =>
   {dly(totalRemainingPower(NL ++ calcTotalRemainingPower(SYSTEM)), roundTime) SYSTEM} .

--- Compute the number of active nodes in the system
op numActiveNodes : Configuration -> Nat [frozen (1)] .
op countNodes     : Configuration -> Nat [frozen (1)] .
--- Compute the percentage of coverage in the system
op coveragePercentage      : Configuration -> Nat [frozen (1)] .
op calcCoveragePercentage  : Bitmap Nat Nat -> Nat .
--- Compute the total remaining power in the system
op calcTotalRemainingPower : Configuration -> Nat [frozen (1)] .


eq numActiveNodes(SYSTEM) = countNodes(extractActiveNodes(SYSTEM)) .

eq countNodes(none) = 0 .
eq countNodes(< O : WSNode | > SYSTEM) = 1 + countNodes(SYSTEM) .


eq coveragePercentage(SYSTEM) =
        calcCoveragePercentage(updateArea(sensingArea, SYSTEM), 0, 0) .

eq calcCoveragePercentage(nil, N, M) = ceiling((N * 100) / M) .
eq calcCoveragePercentage(| nil | BM, N, M) =
        calcCoveragePercentage(BM, N, M) .
eq calcCoveragePercentage(| BIT BITL | BM, N, M) =
if (BIT == f)
then calcCoveragePercentage(| BITL | BM, N, M + 1)
else calcCoveragePercentage(| BITL | BM, N + 1, M + 1)
fi .
```

```
    eq calcTotalRemainingPower(none) = 0 .
    eq calcTotalRemainingPower(< O : RandomNGen | > SYSTEM) =
          calcTotalRemainingPower(SYSTEM) .
    eq calcTotalRemainingPower(MSG SYSTEM) = calcTotalRemainingPower(SYSTEM) .
    eq calcTotalRemainingPower(< O : WSNode | remainingPower : P > SYSTEM) =
P + calcTotalRemainingPower(SYSTEM) .


    --- Generation of a random generator object and a given number of nodes
    op genInitConf : Nat Nat -> Configuration . --- seed numNodes
    op genInitConf : Nat Nat Nat -> Configuration .

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

    ceq genInitConf(M, s(N), N') =
         < L : WSNode | remainingPower : lifetime, status : undecided,
                        neighbors : none, bitmap : initBitmap(L),
        uncoveredCrossings : none,
                        backoffTimer : INF, roundTimer : roundTime,
                        volunteerProb : 1000 / N',
        hasVolunteered : undecided >
                 (if N == 0
                  then < Random : RandomNGen | seed : repeatRandom(M, 3) >
                  else genInitConf(repeatRandom(M, 3), N, N')
                 fi)
if L := random(M) rem sensingAreaSize - (sensingAreaSize / 2) .
                random(random(M)) rem sensingAreaSize - (sensingAreaSize / 2) .


    --- Top left corner of the entire sensing area, which is symetric around (0 . 0)
    op cornerOfSensingArea : -> Location .
    eq cornerOfSensingArea = ((- sensingAreaSize / 2) . (sensingAreaSize / 2)) .

    --- A bitmap of the sensing area of size sensingAreaSize
    op sensingArea     : -> Bitmap .
    op makeAreaColumns : Nat Nat -> Bitmap .
    op makeAreaRows    : Nat -> BitList .

    eq sensingArea = makeAreaColumns(sensingAreaSize / 100, sensingAreaSize / 100) .

    eq makeAreaColumns(N, N') =
         if N > 0 then | makeAreaRows(N') | makeAreaColumns(N - 1, N') else nil fi .

    eq makeAreaRows(N) = if N > 0 then f makeAreaRows(N - 1) else nil fi .

    --- Check if a bitmap is covered by the nodes in the system
    op _coveredBy_ : Bitmap Configuration -> Bool .
    --- Update the bits in a bitmap to t if it is within the sensing range
    --- of an active node in the system
    op updateArea  : Bitmap Configuration -> Bitmap .

    eq BM coveredBy SYSTEM = coverageAreaCovered(updateArea(BM, SYSTEM)) .

    eq updateArea(BM, none) = BM .
    eq updateArea(BM, < O : RandomNGen | > SYSTEM) = updateArea(BM, SYSTEM) .
    eq updateArea(BM, MSG SYSTEM) = updateArea(BM, SYSTEM) .
    eq updateArea(BM, < O : WSNode | status : S > SYSTEM) =
         updateArea((if S == on then updateBitmap(BM, O, cornerOfSensingArea, 100) else BM fi), SYSTEM) .


    --- Check if the system is in the steady state phase
    op steadyStatePhase : Configuration -> Bool [frozen (1)] .

    eq steadyStatePhase(none) = true .
    eq steadyStatePhase(< O : RandomNGen | > SYSTEM) = steadyStatePhase(SYSTEM) .
    eq steadyStatePhase(MSG SYSTEM) = steadyStatePhase(SYSTEM) .
    eq steadyStatePhase(< O : WSNode | status : S > SYSTEM) =
(S =/= undecided) and steadyStatePhase(SYSTEM) .
```

```
    --- Extract the active nodes from the system
    op extractActiveNodes : Configuration -> Configuration [frozen (1)] .

    eq extractActiveNodes(none) = none .
    eq extractActiveNodes(< O : RandomNGen | > SYSTEM) = extractActiveNodes(SYSTEM) .
    eq extractActiveNodes(MSG SYSTEM) = extractActiveNodes(SYSTEM) .
    eq extractActiveNodes(< O : WSNode | status : S > SYSTEM) =
     if S == on then < O : WSNode | status : S > else none fi
     extractActiveNodes(SYSTEM) .


    op sensor-area-covered : -> Prop [ctor] .
    eq { SYSTEM } |= sensor-area-covered =
                  coverageAreaCovered(updateArea(sensingArea, SYSTEM)) .

    op steady-state-phase : -> Prop [ctor] .
    eq { SYSTEM } |= steady-state-phase = steadyStatePhase(SYSTEM) .
endtom)

(set tick max def roundTime .)
---(set tick def 1 .)


**************************************
*****          ANALYSIS:          *****
***** TEST REWRITES AND SEARCHES  *****
**************************************

---  Please notice that the commands take long time to execute
---  and that the output can be large with many nodes. Please consider
---  fewer nodes and fewer rounds for quick results, and consider
---  to pipe the output to file or another program.


--- ACTIVE NODES AND COVERAGE VS DEPLOYED NODES
---(tfrew {genInitConf(1, 1000) dly(activeNodes(nil),roundTime - 1)
           dly(coveragePercentage(nil),roundTime - 1)}
    in time <= roundTime .)

---(tfrew {genInitConf(5, 200) dly(activeNodes(nil),roundTime - 1)
           dly(coveragePercentage(nil),roundTime - 1)}
    in time < roundTime .)


--- COVERAGE AND POWER VS TIME
---(tfrew {genInitConf(313, 75) dly(coveragePercentage(nil),roundTime - 1)
           dly(totalRemainingPower(nil),roundTime - 1)}
    in time < roundTime * 50 .)

---(tfrew {genInitConf(1, 48) dly(coveragePercentage(nil),roundTime - 1)
           dly(totalRemainingPower(nil),roundTime - 1)}
    in time <= roundTime * 50 .)


--- ALPHA LIFETIME VS ALPHA
---(tfrew {genInitConf(47, 75) dly(coveragePercentage(nil), roundTime - 1) }
    in time <= roundTime * 50 .)

---(tfrew {genInitConf(1, 48) dly(coveragePercentage(nil), roundTime - 1) }
    in time <= roundTime * 50 .)


--- ALPHA LIFETIME VS DEPLOYED NODES
--- Note that these take long to execute. Use e.g. 20 nodes and 5 rounds
---   for fairly quick results.
---(tfrew {genInitConf(1, 64) dly(coveragePercentage(nil), roundTime - 1) }
    in time <= roundTime * 200 .)
---(tfrew {genInitConf(45, 80) dly(coveragePercentage(nil), roundTime - 1) }
    in time <= roundTime * 200 .)
---(tfrew {genInitConf(35, 20) dly(coveragePercentage(nil), roundTime - 1) }
```

```
    in time <= roundTime * 200 .)
---(tfrew {genInitConf(8921, 125) dly(coveragePercentage(nil), roundTime - 1) }
    in time <= roundTime * 200 .)


--- STEADY STATE PHASE
---(find latest {genInitConf(73, 7)} =>* { C:Configuration } such that
         steadyStatePhase(C:Configuration)
         in time <= roundTime .)

---(find earliest {genInitConf(791, 5)} =>* { C:Configuration } such that
         steadyStatePhase(C:Configuration) .)

---(mc {genInitConf(2953,5)} |=t (steady-state-phase => [] steady-state-phase)
          in time < roundTime .)


--- STEADY STATE PHASE AND COVERAGE
---(mc {genInitConf(95,5)} |=t [] (steady-state-phase -> sensor-area-covered)
       in time <= roundTime .)
---(mc {genInitConf(5,5)} |=t [] (steady-state-phase -> sensor-area-covered)
       in time < roundTime * 5 .)

---(tsearch [1] {genInitConf(1,5)} =>*
                { < O:Oid : WSNode | status : on, bitmap : BM:Bitmap,
                                     ATTS:AttributeSet > C:Configuration }
    such that BM:Bitmap coveredBy C:Configuration in time <= roundTime .)

---q
```