UNIVERSITY OF OSLO
Department of Informatics

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

Master's thesis

Stian Thorvaldsen

23rd June 2005

Abstract

This master's thesis investigates the suitability of using the real-time formalism and tool Real-Time Maude to formally model and analyze wireless sensor networks.

Real-Time Maude's applicability to wireless sensor networks is explored by formally modeling and analyzing the recently developed wireless sensor network algorithm *the optimal geographical density control algorithm* (OGDC). Wireless sensor networks are a relatively new network domain and pose challenges to the flexibility of existing modeling formalisms. These challenges are met by Real-Time Maude in this thesis, where new aspects, such as power consumption and broadcasting with limited range, are naturally modeled, while integrating other aspects of the OGDC algorithm, like modeling coverage areas and probabilistic behavior without any problems. Additionally, the Real-Time Maude model results in a precise specification of the OGDC algorithm at a fairly high level of abstraction.

The OGDC algorithm has previously been simulated in the network targeted simulation tool *The Network Simulator* (ns-2). In this thesis, I show how analogous simulations can be performed using Real-Time Maude. The results of my simulations deviates somewhat from the results of the simulations using ns-2, for which exact reason was not found (as this is not the focus of this thesis).

Furthermore, I show how Real-Time Maude's other formal analysis capabilities complement simulation tools, by subjecting the OGDC algorithm to state exploration analysis which investigates *all* behaviors in the system from a given initial state, whereas simulations investigates only *one* behavior. This analysis is performed on a few nodes, because of the large state space. State space exploration is performed for several chosen initial states in search for design errors in the protocol, but no significant errors were found.

Acknowledgments

I would like to thank my supervisor Peter C. Ölveczky for good guidance, helpful advice, and his efficient red pen. I also want to thank Jennifer C. Hou at the University of Illinois at Urbana-Champaign for suggesting the OGDC algorithm to us, which I found both challenging and a delight to model and analyze.

I also want to thank my friends for all the fun lunch breaks at the university and all the good times outside. Thanks to Mamma and Pappa for making it a joy to live at home. Finally, a special thanks to Jen for her patience and understanding, and everything else.

Contents

1 Introduction **1**

2 The Optimal Geographical Density Control Algorithm **4**
- 2.1 Overview of the OGDC Algorithm . . . 5
  - 2.1.1 High-level description . . . 5
  - 2.1.2 The optimal position . . . 7
  - 2.1.3 The starting nodes . . . 9
  - 2.1.4 Receiving power-on messages . . . 10
  - 2.1.5 Relaxing the assumptions . . . 12

3 Real-Time Maude **13**
- 3.1 Object-oriented specification in Maude . . . 13
- 3.2 Object-oriented specification in Real-Time Maude . . . 14
- 3.3 Simulation and formal analysis in Real-Time Maude . . . 15

4 The Real-Time Maude Model of the OGDC Algorithm **17**
- 4.1 Some modeling challenges . . . 17
- 4.2 Ambiguities and implicit assumptions in the original specification . . . 18
- 4.3 Modeling time and time elapse . . . 19
- 4.4 The definition of a node . . . 20
- 4.5 Modeling communication . . . 23
- 4.6 Random number generation . . . 25
- 4.7 The node's coverage area . . . 26
  - 4.7.1 Initializing the bitmap . . . 27
  - 4.7.2 Updating the bitmap . . . 28
- 4.8 Computing the coverage area crossings . . . 29
- 4.9 Computing the backoff timers . . . 30
- 4.10 The OGDC algorithm . . . 31
  - 4.10.1 The volunteering process . . . 33
  - 4.10.2 Ignoring and receiving power-on messages . . . 34
  - 4.10.3 The activation of a node . . . 36
  - 4.10.4 Death of a node and restarting the round . . . 37

5 Analysis of the OGDC Algorithm **39**
- 5.1 The analysis environment . . . 40
  - 5.1.1 The sensing area . . . 40
  - 5.1.2 Generating initial states . . . 40
  - 5.1.3 The time sampling strategy . . . 43
- 5.2 The ns-2 simulations of the OGDC algorithm in Real-Time Maude . . . 43

5.2.1 The number of active nodes and percentage of coverage with respect to the number of deployed nodes 45
5.2.2 The percentage of coverage and total amount of remaining power with respect to time 48
5.2.3 α-lifetime with respect to α and with respect to the number of deployed nodes 50
5.2.4 Iterative simulations using Real-Time Maude's meta-level 52
5.3 Further analysis of the OGDC algorithm in Real-Time Maude 52
5.3.1 Reaching the steady state phase 52
5.3.2 Coverage 54
5.3.3 A node's status and coverage area 55
5.3.4 When nodes die 57
5.4 Concluding remarks of the analysis 58

6 Conclusion **59**

Bibliography **60**

A The Real-Time Maude specification of the OGDC algorithm **63**

Chapter 1

Introduction

With the development of new technology, the areas to which computer networks can be applied are constantly growing. The recent advances in micro-technology has made it possible to develop small, low-power, and cheap computers, called *sensor nodes*, see e.g., [22]. These nodes are battery operated, and, because of their size, have a limited amount of computer memory and limited computational power. They are equipped with sensing technology which enables them to observe different aspects of their surrounding environment, such as motion, heat, sound, etc. They are also equipped with wireless communication technology which makes them able to communicate with each other to form a new kind of networks, *wireless sensor networks*.

The concept of such a network has opened up a new range of applicable areas where environment observation and computer networking is desirable. Such areas can be battlefield surveillance for targeting of enemy forces, or more socially useful areas as monitoring movements in the ocean in search for warning signs of approaching tidal waves, or measuring the condition of soil to achieve better crop by providing it with the right nourishment. In the future, the nodes may become sufficiently small to fit in the bloodstream of human beings, to monitor health conditions and discover health risks at an early stage. Other applicable areas are mentioned in [1].

Since the nodes in a wireless sensor network have a limited power supply, in the form of a battery, and are possibly deployed in areas that are hard to reach, like in the middle of a forest fire, can make power replacement virtually impossible. Consequently, a wireless sensor network has a limited lifetime. To increase the lifetime of the network, more nodes than strictly required to observe a certain area, the *sensing area*, are deployed. The higher density of nodes allows for some nodes to temporarily go to "sleep" to conserve power, while others perform the *sensing task*.

Because of the application areas of wireless sensor networks, the nodes communicate by a wireless medium, of which there are mainly three types: infrared, optical and radio. Among these, radio is the most commonly used, because it does not rely on clear line of sight. The nodes, therefore, use broadcast as form of communication. Because of the nodes' limited power supply, and the physical constraints, the sensor nodes are equipped with small radio transmitters, which limits the transmission range of the broadcast. The nodes can then propagate the information generated from their observations through the network, and often to a common node, *the sink*, which sends the information to the end-user(s).

A new type of computer networks often introduces untraditional aspects which can be hard for existing network protocols to capture. A wireless sensor network receives virtually no administration after deployment, and it is therefore desirable that the network can observe the entire sensing area for as long time as possible. This, in combination with the fact that dying nodes may results in time consuming re-routing of information and re-organization of the network,

raises a demand for *power-aware* protocols in wireless sensor networks. Power consumption has not been a main concern of existing communication protocols, and traditional wireless network protocols are usually based on other forms of communication than broadcast. The emergence of wireless sensor networks, therefore, causes a new range of communication protocols to be developed, with focus on these new aspects. Furthermore, in the development of a new research field, the demand for formal modeling and analysis of such a field increases. Prototyping new sophisticated protocols in actual wireless sensor networks can both be difficult and time consuming, as the wireless sensor network is a relatively new field of research for which good testbeds may not yet have been created. Formal modeling and analysis of wireless sensor networks and its protocols can, therefore, be both time and cost economic, as formal modeling and analysis can be performed while new testbeds are created, to possibly discover critical design errors in protocols in early stages of development.

Another consequence of a new field of network research is that existing tools, targeted at "ordinary" network and communication protocols, may not easily support the new aspects, or combination of aspects, introduced by the new domain. Specialized network tools are often optimized for certain types of networks and/or selected forms of communication, and their formalisms may be too restricted to capture the new aspects in a natural way. Wireless sensor networks, therefore, pose a challenge to formalisms and tools that have traditionally been used to simulate and analyze network protocols based on a different communication form, or where power consumption has not been a main concern.

Real-Time Maude [15, 19] is a formalism and a simulation and analysis tool for real-time systems. It has a flexible specification formalism, with a natural integration of data types, functional aspects and dynamic behavior. It also has a wide range of analysis capabilities, including simulations of single behaviors and exhaustive state space exploration strategies. The main focus of the formalism is logical clarity and good support for object-oriented specification of real-time systems. Real-Time Maude covers the "middle ground" between informal specifications on the one hand, and specialized simulation tools, like ns-2 [14], GloMoSim [24], and JiST [2], on the other. Whereas an informal specification is often non-executable and ambiguous, a Real-Time Maude specification is both precise and executable. Specialized network targeted tools often provide support for selected types of networks and communication protocols. Real-Time Maude is not specially designed for any particular type of networks and is not based on any fixed form of communication or class of protocols. The network's different aspects, like the communication primitives, are left up to the user to define.

With Real-Time Maude's general formalism and possibilities for high level modeling, an executable specification can be relatively quickly modeled. This is desirable with respect to a fast growing research field like wireless sensor networks, where the development of domain-specific tools may have a hard time keeping up. The specification can then be subjected to formal analysis, in addition to simulations, with search and model checking, where possible design errors can be uncovered in an early stage of development. Real-Time Maude has previously been successfully used to model and analyze other complicated multicast protocols [18, 9], where subtle, but significant, errors have been discovered which were not found during extensive simulations.

Given the flexibility of the Real-Time Maude formalism and its support for object-oriented modeling of real-time systems, we conjecture that it should be a good candidate to formally model and analyze wireless sensor network protocols. Given Real-Time Maude's possibilities for high level of abstraction, the resulting model should offer a precise formal protocol specification without unnecessary details, which is *relatively* intuitive to understand. Real-Time Maude's analytical capabilities should be able to offer complementing formal analysis to the single behaviors that are simulated in specialized simulation tools, in search for design errors. In this thesis we wish to investigate the suitability of using Real-Time Maude to model and analyze today's

new sophisticated wireless sensor network protocols.

Jennifer C. Hou, at the University of Illinois at Urbana-Champaign, suggested to us a recently developed sophisticated wireless sensor network algorithm, developed by Honghai Zhang and Hou, as a challenging modeling and analysis task for the real-time system formalism and tool Real-Time Maude. The algorithm captures several of the aforementioned new aspects introduced by wireless sensor networks. *The optimal geographical density control algorithm* (OGDC) [25] is a power-aware algorithm, whose goal is to maintain complete sensing coverage and connectivity for as long time as possible. It has been simulated in the simulation tool *The Network Simulator* (ns-2) [14] and the performance was compared to other similar algorithms with good results.

How can Real-Time Maude handle this challenge, and to what degree is it suitable for modeling and analyzing this new kind of algorithm in a new application domain? This thesis is about the modeling and analysis effort of the optimal geographical density control algorithm.

The rest of the thesis is organized as follows. Section 2 gives an description of the different aspects of the OGDC algorithm. Section 3 gives an introduction to Real-Time Maude's formalism and analysis capabilities. In Section 4, the formal modeling of the OGDC algorithm is outlined. The formal analysis of the OGDC algorithm follows in Section 5. Here it is explained how all the simulations, corresponding to those performed in [25] by Zhang and Hou using ns-2 with the CMU wireless extension, are done in Real-Time Maude. In addition, we explore all possible behaviors from certain initial states by search, model checking, and time-specific search. The thesis is concluded in Section 6.

The full specification of the OGDC algorithm is listed in Appendix A. The specification will also, hopefully, be available at Real-Time Maude's homepage [19], so that anybody can experiment with it.

Chapter 2

The Optimal Geographical Density Control Algorithm

A wireless sensor network usually consists of a large number of sensor nodes with limited battery capacity. The sheer number of nodes, which may be unreachable, makes power replacement virtually impossible. It is therefore important that the 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 sensing area. Connectivity means that each node in the network can reach all of the other nodes. The nodes are often deployed with virtually no information about their surroundings. That is, a node has no knowledge of the position of other nodes or their condition. Most tasks must therefore be performed based on the information gathered after deployment.

The reason for deploying a large number of nodes in a wireless sensor network is that the network can tolerate that not all the nodes are *active* all the time, so the *inactive* nodes can save power. A node is inactive when it is dead, either from lack of power or because it is physically destroyed, or when it is intentionally switched off because its presence is not needed. A node that is switched off can of course be switched on when needed. The process choosing the nodes that can be switched off is called the *density control process*.

The *optimal geographical density control* algorithm (OGDC) [25] is a recently published density control algorithm, developed by Honghai Zhang and Jennifer C. Hou at the Department of Computer Science, University of Illinois at Urbana-Champaign. The OGDC algorithm is a completely localized density control algorithm, whose goal is to maintain complete coverage and connectivity of a wireless sensor network for as long time as possible. The only way to prolong the lifetime of a wireless sensor network is to make some nodes temporarily inactive to conserve their power, since an inactive node consumes significantly less power than an active node. The OGDC algorithm is therefore a sophisticated algorithm for periodically selecting nodes to be active and inactive in order to maximize the lifetime of the network. In addition, the OGDC algorithm is a *localized* algorithm in the sense that each node uses only local information to carry out the density control process.

The active nodes are carefully chosen based on the position of other neighboring nodes that are already active. In the density control process, only the nodes that contribute "maximum additional coverage" to the existing active nodes are chosen to be active nodes themselves. The density of the nodes in a sensor network is often high, so the number of neighbors a node has to take into consideration can be quite large. The set of active nodes also varies with time to achieve as long network lifetime as possible.

Several means of communication can be used in wireless sensor networks, depending on, among other things, the physical capabilities of the nodes and their environment. The most

commonly used form of communication is radio transmission, which is also assumed in OGDC. The method of communication in OGDC is therefore broadcasting. The broadcast works with *limited* signal strength, and therefore has *limited* transmission range. This means that only nodes that are within a given distance from the sending node will be able to receive the broadcast.

Time plays a crucial role in the OGDC algorithm. First of all, OGDC is a real-time algorithm in that the nodes have to be *time synchronized*. That is, all nodes must agree on when "the beginning of time" is. The reason for this is that several rounds of OGDC are run within the lifetime of the network. In order for the nodes to be able to start each new round of the algorithm at the same time, they need to be time synchronized from the first round of OGDC, and need to remain time synchronized throughout the lifetime of the network as well. The choice of active nodes is also heavily based on time, where *backoff timers* are used to decide *when* the nodes should perform certain actions. Furthermore, because of the wireless communication capacity of radio transmission, a broadcast from a node takes some time to reach its destination(s). Broadcasting a message by radio transmission takes time in the magnitude of milliseconds, whereas the lifetime of the sensor network is in the magnitude of hundreds of thousands of seconds. These small "time steps" together with the long lifetime, leads to tens of millions of "time steps" during the lifetime of the network.

Section 2.1.1 gives a high-level overview of the OGDC algorithm. In [25], the authors make the following reasonable assumptions to focus on the central parts of the algorithm:

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

Ensuring that each node is aware of its relative or geographical position is known as the *localization problem* [4, 20, 5]. This is a separate field of research in the wireless sensor networks community, and falls outside of the scope of the OGDC algorithm. We may assume that some sort of localization protocol has already been used to make sure that each node is aware of its own position.

It is proved in [25] 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.

The time synchrony assumption lets us work on a network where all the nodes are time synchronized from the first round of OGDC. A great deal of research is done on time synchronization as well [11, 21, 10], and we may assume that a time synchronization protocol used prior to OGDC.

The paper [25] also devises ways to relax the two last assumptions, they are outlined in Section 2.1.5. Zhang and Hou extend the OGDC algorithm to handle *k-coverage*, which is defined to that the sensing area is only covered if each point is covered by at least k nodes. I will, however, model my Real-Time Maude specification as 1-coverage under the above assumptions, so that I can focus on the actual analysis of this model, rather than on doing the necessary extensions needed to relax these assumptions.

2.1 Overview of the OGDC Algorithm

2.1.1 High-level description

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

- the node selection phase, and

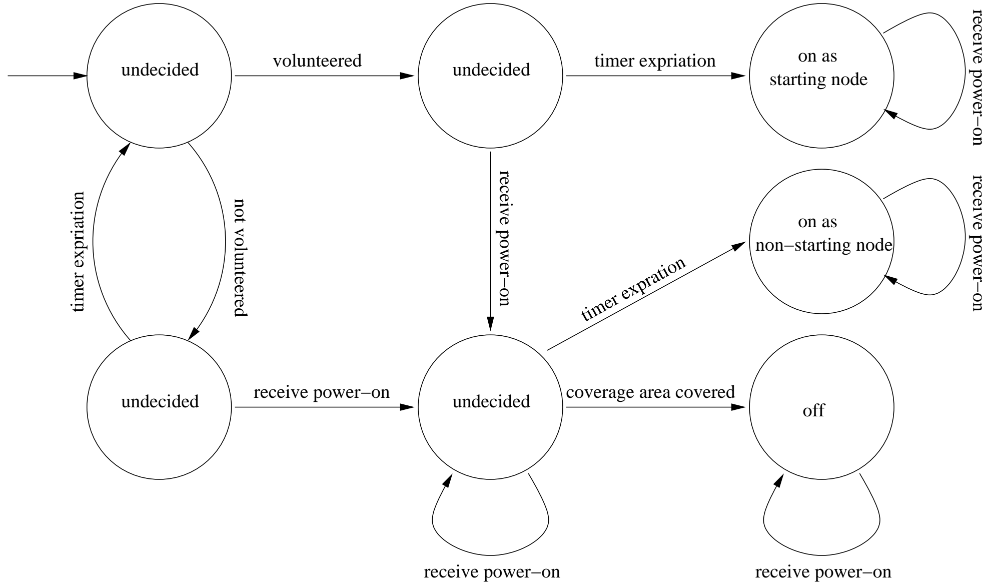


Figure 2.1: A high-level description of the stages each node goes through in OGDC.

- the steady state phase.

The *node selection phase* starts at the beginning of each round of the OGDC algorithm, where the set of active nodes is selected. In this phase, each node is in one of three states[1], "undecided", "on" or "off". The *steady state phase* begins when each node is either in the "on" or "off" state, and the network is stable and can perform its sensing task.

The node selection phase starts with a *volunteering process* where each node probabilistically decides whether to volunteer to be a *starting node* (see Figure 2.1). Each node that volunteers sets its *backoff timer*. A volunteered node becomes a starting node when its backoff timer expires. It is then the "on" state and broadcasts a *power-on* message. The backoff timer aims at preventing that several volunteered neighboring nodes become starting nodes. If a volunteered node receives a power-on message within the expiration of its backoff timer, the backoff timer is reset and the node does not become a starting node. Each node that does not volunteer sets its backoff timer to a larger value. If it does not receive a power-on message within the expiration of its backoff timer, it repeats the volunteering process, with the volunteering probability *doubled*. This ensures that a node which is not within radio transmission range of any other nodes, a situation that may arise when nodes start dying, eventually becomes a starting node, and consequently an active node, because the node is guaranteed to volunteer as a starting node when the volunteering probability reaches 1.

When a node receives a power-on message, it (re)sets its backoff timer to a value computed from a set of conditions, as outlined in Section 2.1.4. The computation of the backoff timer ensures that the node whose presence is most beneficial to the network at that time gets the lowest valued backoff timer. Which node's presence benefits the network most, changes as nodes become active. The node which gets the lowest timer is the node which is *closest* to the *optimal position*, as explained in Section 2.1.2. its backoff timer expires, the node enters the "on" state and broadcasts a power-on message. It then gets a chance to influence the selection of the rest of the active nodes. Therefore, only the nodes that benefit the network the most at any given time become active nodes.

[1]In the Real-Time Maude model, this is called the node's *status*.

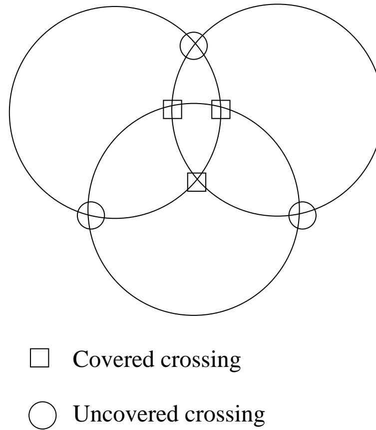


Figure 2.2: Crossings that are not covered by a third node is uncovered.

If a node has its entire coverage area[2] covered by its neighbors, it sets its state to "off". When each node has set its state to either "on" or "off", the network enters the steady state phase, with the nodes with state "on" as the set of active nodes. When a round is over, each node resets its state back to "undecided", and the volunteering process starts over again.

2.1.2 The optimal position

The reason for deducing the concept of an *optimal position* in [25] is to prolong the operational lifetime of the network by minimize the number of active nodes. This is done by only choosing nodes that are *closest* to the optimal positions to be part of the set of active nodes. The optimal position is the position where an active node should be located to be most beneficial to the network. That is, the position where the node contributes maximum additional coverage to the existing set of active nodes, while still staying connected.

Each node that becomes active broadcasts a power-on message, and the optimal position is computed by each node that receives the message, based on the information about its *neighbors*[3]. Depending on the position of its neighbors, the optimal position is located either with respect to

- an *uncovered crossing* or
- a single node.

The intersection of the boundaries of two active nodes' coverage areas is called a *crossing* (see Figure 2.2). This crossing is *uncovered* if it is not within the coverage area a third active node. The receiving node's timer is then set to a value that is proportional to the distance it is from the optimal position. This ensures that the closer the node is to the optimal position, the earlier it becomes an active node.

A requirement of an operational sensor network is to provide both coverage and connectivity. The assumption that the radio transmission range is at least twice as large as the sensing range allows us to only worry about coverage. According to the following lemma in [25], coverage is guaranteed if there are no uncovered crossings in the entire sensing area:

[2]The coverage area of a node will denote the area which is within the sensing range of the node.

[3]The word *neighbor* will have the following meaning: node B is node A's neighbor if A has received a power-on message from B.

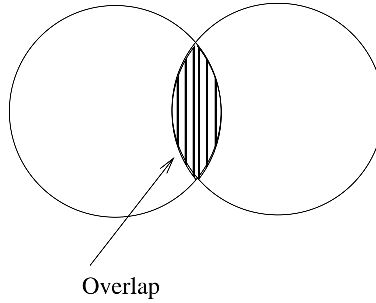


Figure 2.3: Overlap is the intersecting area of two active nodes.

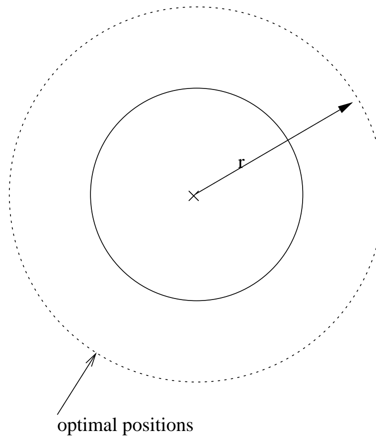


Figure 2.4: The optimal position with respect to a single node.

Lemma 1 *If there are no uncovered crossings in the sensing area, the entire sensing area is covered by the set of active nodes.*

By choosing nodes that cover the uncovered crossings to be active nodes, the entire sensing area is covered. To minimize the number of nodes in the set of active nodes, another lemma is devised in the same paper:

Lemma 2 *Minimizing the number of active nodes is the same as minimizing the* overlap *of the nodes' coverage areas.*

The overlap is the intersecting coverage areas of two active nodes (see Figure 2.3). The goal of OGDC, to prolong the networks operational lifetime, is therefore achieved by choosing the set of active nodes such that they provide the minimum amount of overlap while leaving no crossing uncovered.

Depending on the location of the receiving node's neighbors, the node sets its backoff timer either depending on how close it is to cover an uncovered crossing with the minimum amount of overlap, or to provide the best basis for that to happen for future active nodes, if there are no uncovered crossings within its coverage area.

The exact location of the optimal position in the latter case is computed with respect to a single neighbor. The optimal position is, with respect to a single neighbor, anywhere on a circle with center in the location of the neighbor and a fixed radius r (see Figure 2.4). The radius r has a value that minimizes overlap between these two nodes and the future active node that

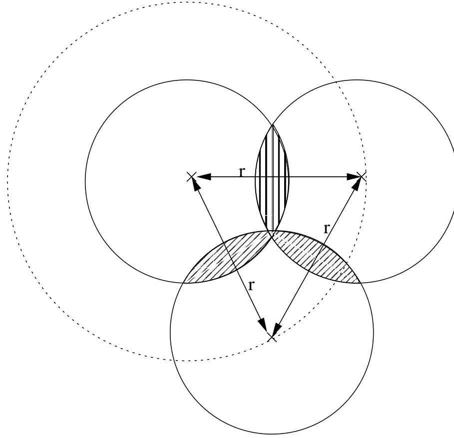


Figure 2.5: The optimal relative locations for three nodes.

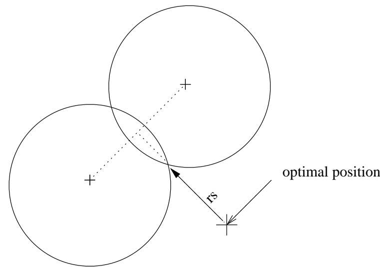


Figure 2.6: The optimal position with respect to an uncovered crossing. rs is the radius of a node's coverage area.

covers the crossing of the two nodes (see Figure 2.5). Since the optimal position is anywhere on the circle, a representative for the optimal position is chosen on random, as explained in Section 2.1.3, so that the receiving nodes can compute how close they are to the point and set their backoff timers accordingly.

If the receiving node has an uncovered crossing within its coverage area, the optimal position is on a line from the crossing, perpendicular to the line between the two nodes that creates the crossing. The optimal position is on this line so that the crossing is just within the coverage area of the node (see Figure 2.6).

By constantly choosing nodes that "best" covers uncovered crossings or provide this basis for other future active nodes, the whole sensing area is covered according to Lemma 1, and it creates the minimum amount of overlap, and thereby minimizing the number of active nodes according to Lemma 2. With a constant minimum set of active nodes in the network, the lifetime of the network will be maximized.

2.1.3 The starting nodes

The starting nodes are the ones that kick off the process of selecting active nodes. It is desirable to have only a few starting nodes. That way, more active nodes are chosen based on their positions with respect to the existing active nodes, instead of nodes starting up their own. The volunteer probability is, therefore, set to a value that, statistically, results in one node volunteering to be a starting node in a round. There is also a minimum power requirement,

called *power threshold*, that has to be fulfilled for a node to be able to volunteer as a starting node. The power threshold is set to a value that ensures with high probability that the node can remain active through the whole round. Why this is important is explained in Section 5.3.3.

The power-on message broadcast by the starting node, when it enters the "on" state, contains the *location of the node* and a *random direction*. The random direction denotes where on the circle with radius r, mentioned in Section 2.1.1, the representative for the optimal position is (see Figure 2.4).

2.1.4 Receiving power-on messages

A node influences the selection of other active nodes when it becomes active and broadcasts a power-on message. Each node that receives this power-on message takes it into consideration and possibly resets its existing backoff timer. The new value is based on how close it is to the optimal position, computed from the list of neighbors which is updated with the node that sent the power-on message.

When a node receives a power-on message, and is not in the "on" state or dead, it checks whether its entire coverage area is covered by its neighbors. If this is the case, the node sets its state to "off". How to check whether a coverage area is covered by other nodes, is discussed in Section 4.7. If the node's coverage area is not covered, it computes its backoff timer depending on the following conditions:

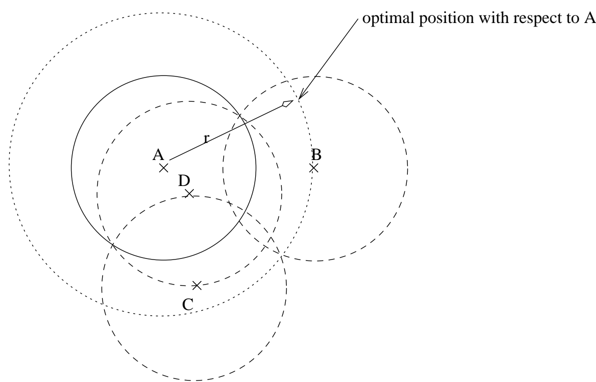
a) There exists an uncovered crossing within the coverage area of the node.

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

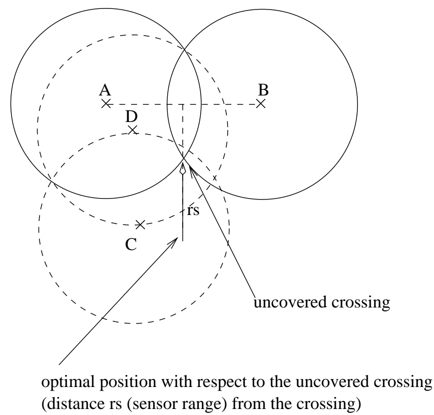
c) There exists no uncovered crossings nor starting neighbors.

If c is true it sets it backoff timer to T_c, which is a constant. This constant is large enough to provide the node a reasonable amount of time to receive more power-on messages before the backoff timer expires. If the node receives power-on messages that results in b becoming true, it resets its backoff timer to T_b which is a value proportional to how close it is to the optimal position with respect to the starting neighbor (see Figure 2.4). If a node receives even more power-on messages causing a to be true, it resets its backoff timer to T_a which is a value proportional to how close it is to the optimal position with respect to the uncovered crossing (see Figure 2.6). If the node is close to an optimal position, it gets a significantly smaller backoff timer, which expires quickly, and the node becomes active. The node then broadcasts a power-on message and can thereby influence the selection of other active nodes, because they might reset their backoff timers based on this node becoming active. The power-on message sent contains -1 in the direction field. In this way the receiver of the power-on message can tell whether the power-on message came from a starting node, to be able to determine the validity of statement b.

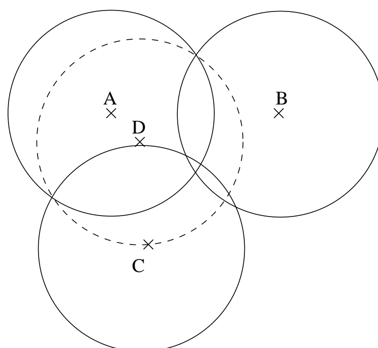
This section ends with an example: In Figure 4.2, node A volunteers as a starting node. When its backoff timer expires it broadcasts a power-on message with a random direction, which is received by the nodes B, C and D in Figure 2.7(a) Since B is closer to the optimal position, the point on the circle with radius r determined by the random direction, with respect to A, the value of B's backoff timer is smaller than both the values of C's and D's backoff timers. This results in B becoming active first in Figure 2.7(b). When B becomes active it broadcasts a power-on message which is received by A, C and D. A ignores this power-on message since it is already active. C and D reset their existing backoff timers to new values, because they both have uncovered crossings within their coverage area as a result of B becoming active. C gets a smaller valued backoff timer than D because it is closer to optimal position with respect to



(a) Node A becomes active and broadcasts a power-on message. Node B, C and D compute their backoff timers based on distance to the optimal position with respect to node A.



(b) Node B becomes active and broadcasts a power-on message. Node C and D compute their backoff timers based on their distance to the optimal position with respect to the uncovered crossing.



(c) Node C becomes active and broadcasts a power-on message. Node D's coverage area is completely covered and it can be switched off.

Figure 2.7: Optimal positions with respect to a starting node, and an uncovered crossing

the uncovered crossing. In Figure 2.7(c) C becomes active and broadcasts a power-on message, which both A and B ignores, but is received by D. D discovers that its entire coverage area is covered by A, B and D, and is therefore switched off.

2.1.5 Relaxing the assumptions

The last two of the three assumptions are suggested loosened in [25] as outlined here:

The radio range is less than twice the sensing range

The problem that arises here is that coverage no longer implies connectivity. The only thing this changes, is that more elaborate tests are needed to establish when a node can safely be put in the "off" state. It is no longer sufficient for its coverage area to covered. Before a node can be put in the "off" state, it also has to ensure that the network remains connected after it is switched off.

Time synchronization

The time synchronization issue is suggested solved, as assumed in Section 2, by having an initial time synchronization round. A few nodes are synchronized before deployment and add a field in the power-up message that is broadcast, which states at what exact time the next round of OGDC should start. A non-starting node should reduce this value with the amount of time elapsed since it received the last power-on message, and put this value in its broadcast power-on message. This way, all the nodes should start up at the same time in the next round.

Chapter 3

Real-Time Maude

Real-Time Maude [15, 19] is a high-level declarative language and tool for formal specification and analysis of real-time systems, and extends Maude [6, 7] by adding real-time features. The focus of Real-Time Maude is expressiveness and ease of specification, and it is particularly well suited for specification of object-oriented real time systems. Real-Time Maude is efficiently implemented, based on an internal translation to Maude code which is processed by the underlying Maude rewrite engine. The specification formalism is based on *real-time rewrite theories* [17], which extend *rewriting logic* [13, 3], on which Maude's specification formalism is based. We first look at how specification is done in Maude, and then outline the Real-Time Maude extensions.

3.1 Object-oriented specification in Maude

A Maude module is an algebraic specification based on a rewrite theory (Σ, E, R), where Σ is the module signature, E is a set of equations, and R is a set of labeled conditional rewrite rules. The equations E specify the static aspects of the system and together with Σ define an equational logic theory, that specifies the state space of the system. The rewrite rules R specify the system's dynamic transitions, and are logically used *modulo* the equations. Operationally, this means that each term is reduced to its canonical form by the equations, before a rule is applied. Each rewrite rules has one of the forms:

```
rl [l] : t => t' .
crl [l] : t => t' if cond .
```

which is understood as the term t is rewritten to the term t' by rule l. In the conditional rule (`crl`) the rewrite can only happen if the condition *cond* evaluates to `true`. The label l can be considered as a sort of comment. A rewrite theory defines all possible behaviors of a system.

Object-oriented modules are specified with the syntax `(omod ... endom)`. Inside the module, sorts (`sort ...`), subsorts (`subsort ...`), and operators can be declared. Operators define each sort's respective domain of terms (`op ...`), and the '_' denotes the position(s) of the argument(s). The operator can be declared with attributes, such as `assoc`, `comm`, and `id` that declare the operator to be, respectively, associative, commutative, and with an identity element, which are used in the matching of terms. Equations are introduced with the keyword `eq`, or `ceq` for conditional equations. Variables, which are used in equations and rules, are declared with the syntax `var`.

A class C, of which objects can be instantiated, have the following syntax:

```
class C | a1 : s1, a2 : s2, ... , an : sn .
```

where each a_i is an attribute of sort s_i. A message m, with parameters of sorts $s_1 ... s_n$, is declared as follows:

msg m : s_1 ... s_n $\rightarrow$ Msg .

The rules specify the dynamic behavior of objects. They rewrite terms of sort `Configuration`, which is a multiset of objects and messages. The rules are used from left to right, so that the rule:

```
rl [l] : < O : C | a1 : v1, a2 : v2 > =>
         < O : C | a1 : v1 - 1, a2 : v2 > m(v2) .
```

specifies a transition in which an object with identifier `O` and attribute values `v1` and `v2`, produces a message `m` with parameter `v2`, and decreases the value of its `a1` attribute. Attributes that are not altered by the rule, like `v2` in this case, can be omitted from right-hand sides of rules, and attributes that are not altered and do not affect other attributes can be omitted completely.

3.2 Object-oriented specification in Real-Time Maude

Real-Time Maude is based on *real-time rewrite theories* which contain:

- A sort `Time` as the time domain. The time domain can be user specified, but Real-Time Maude provides built-in modules for useful time domains as `NAT-TIME-DOMAIN-WITH-INF` (the natural numbers with an "infinite" value `INF`) and `POSRAT-TIME-DOMAIN-WITH-INF` (the nonnegative rational numbers with `INF`).

- "Tick" rewrite rules which advance time in the system. The time must advance uniformly, so that time elapses at the same "pace" in all parts of the system. To achieve this, a built-in constructor `{_}` of the new sort `GlobalSystem`, which capture the *whole* state of the system, is used together with tick rules which must be on the form:

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

  which rewrites the term `{`t`}` to `{`t'`}` if *cond* evaluates to `true`, where u of sort `Time` denotes the "duration" of the rewrite. The initial state of the system should always have the form `{`t_0`}` to ensure uniform time elapse. Ordinary rewrite rules are considered as instantaneous transitions which are assumed to take zero time.

For modeling time elapse in an object-oriented systems, the tick rule is usually [16]:

`crl [`l`] : {`t`} => {delta(`t`, `R`)} in time` R `if` R `<= mte(`t`)`

where R is a variable of sort `Time`, and `delta` is a function that defines the effect of time elapse on the system, and `mte` is a function that defines the maximum amount of time that can elapse in the system, before something "important" (such as the application of an instantaneous rule) must happen.

3.3 Simulation and formal analysis in Real-Time Maude

Real-Time Maude extends the simulation and state exploration search strategies of Maude by offering *timed* fair rewriting, *timed* search and *timed* linear temporal logic model checking. Additionally, Real-Time Maude offers two new time-specific search commands.

Simulation in Real-Time Maude is performed with timed fair rewriting command:

```
(tfrew init in time <= R .)
```

where *init* is the initial state, and R is a term of sort `Time` which denotes the *time bound* for the rewrite. That is, the simulation is run until the time bound is reached. Timed fair rewrite executes *one* behavior in the system, and is useful for prototyping and simulation purposes.

Timed search uses a breath-first strategy to investigate *all* possible behaviors that can be reached from the initial state:

```
(tsearch init =>* pattern such that cond in time <= R .)
(tsearch init =>* pattern such that cond in time-interval between <= R and >= R' .)
```

which searches for states matching *pattern* that satisfy the condition *cond*. R and R' are of sort `Time`, and `=>*` and the time bound can be replaced as explained in [15].

Real-Time Maude offers two new time-specific searches:

```
(find latest init =>* pattern such that cond in time <= R .)
(find earliest init =>* pattern such that cond .)
```

which finds the state that matches *pattern* and satisfies *cond*, that took the longest and the shortest time to reach (for the first time) from the initial state, respectively.

Furthermore, Real-Time Maude extends Maude's efficient *linear temporal logic model checker* to check whether each behavior within a time bound satisfies a temporal logical formula. A state proposition *prop* is defined as follows:

```
(eq {pattern} |= prop = b .)
```

for b a term of sort `Bool`. The state proposition *prop* holds in all states where *prop* evaluates to `true`. A temporal logic formula is then specified from the set of state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\` (conjunction), `->` (implication), `[]` (always), `<>` (eventually), `U` (until), etc. To check if a temporal logic formula holds in all behaviors within time R we issue the command:

```
(mc {t} |=t formula in time <= R .)
```

The model check command returns `true` if the temporal logic formula holds in all behaviors in the system starting from the initial state. If the formula does not hold, a counterexample is returned, in form a path to a state where the formula is `false`.

Finally, Real-Time Maude also provides "untimed" search and model checking, which can be used when the state space reachable from the initial state is finite.

Real-Time Maude is parametric on the time domain which means we can specify *dense time* (using, e.g., the nonnegative rational numbers). In such case, the analysis may be "incomplete", because the analysis is done with respect to the *time sampling strategy*, which may "skip" essential

points in time, where new behaviors can originate. The analysis in dense time is, therefore, performed with respect to the time sampling strategy, which is chosen by the user. For example, if the time sampling strategy is defined to advance time with time steps of size T and the application of an instantaneous rule l at time $t + T/2$ results in a new behavior, this behavior can not be investigated because the time sampling strategy "skips" this point in time. However, the application of instantaneous rules are often triggered by an event, like in the specification of the OGDC algorithm where each instantaneous rule is triggered either by the expiration of a timer or by the arrival of a message. In this case, dense time does not cause any difficulties. The "important" points in time are defined by the `mte` function, which defines the next point in time where something "important" must happen.

Chapter 4

The Real-Time Maude Model of the OGDC Algorithm

This chapter gives an overview of the Real-Time Maude model of the OGDC algorithm. Real-Time Maude's general specification formalism allows a wide range of modeling possibilities for the different aspects of the OGDC algorithm. Active use of modules, and well-defined module interfaces, allowed for separate prototyping of each module. This was particularly useful in the early stages of the modeling process, where the formalization of the OGDC algorithm revealed several implicit assumptions, which led to small but significant changes in the model. Most ambiguities and implicit assumptions were easily resolved and some needed more attention, but they are clarified and made explicit in the Real-Time Maude model.

Section 4.1 lists some of the modeling challenges wireless sensor networks and the OGDC algorithm pose to modeling formalisms and analysis tools. Section 4.2 sums up the most significant ambiguities and implicit assumptions, and the decisions we made in their respect. Section 4.3 outlines how time and the effect of time are modeled in Real-Time Maude, followed by definitions of nodes and message in Sections 4.4 and 4.5. In Section 4.6 we define how random numbers are generated. Section 4.7 explains how the bitmap for a node's coverage area's is handled. Section 4.8 and 4.9 outlines how the crossings and backofftimers are computed, respectively. Finally, Section 4.10 contains the Real-Time Maude rules that define the OGDC algorithm.

4.1 Some modeling challenges

Wireless sensor networks and the OGDC algorithm pose many challenges to modeling formalisms and analysis tools:

- The topology in a wireless sensor network is rapidly changing in that nodes are switched on and off. In addition, a node may die from lack of power resulting in a constant changing network topology.
- Power consumption is the most critical part of the OGDC algorithm, and has to be faithfully modeled, such that each node consumes different amounts of power with respect to its status and actions.
- The form of communication is broadcast with limited range, without the nodes having any knowledge of surrounding topology. That is, a broadcasting node is unaware of the *number* of nodes which are within its transmission range, and has no knowledge of *which* nodes that are within range.

- The broadcast is subject to *transmission delays* because of the limited capacity of wireless transmission. In addition, each node's behavior is entirely dependent on the expiration of timers. Both delays and timers need to be adequately modeled. The value of a node's backoff timer is based on relative angles and distances between the node and its neighbors, which need to be accurately computed.
- Each node has a fixed sensing range and is responsible to keep track of its own coverage area, which involves modeling of geometric areas.
- Several aspects of OGDC demands "random" or probabilistic behavior.
- The large number of nodes in the network also poses a challenge to the analytical capabilities of analysis tools.

This is a diverse list of challenges involving an ad-hoc network topology, resource consumption, broadcast communication with limited range, time-sensitive behavior, area coverage, and probabilistic behavior. Specialized formal specification and analysis tools can offer powerful simulation and analytical features for selected areas. However, their formalisms can sometimes be too restrictive to support algorithms with diverse requirements, such as the OGDC algorithm. With the birth of a new research area, such as the field of wireless sensor networks, new forms of algorithms are devised which demand rapid prototyping in the early stages of development. For the previously mentioned specialized tools to be able to support these new algorithms, extensions, or new tools all together, may have to be developed. However, the prototyping and further analysis of the actual algorithms would have to wait for the completion of these tools.

This chapter describes how the expressive formalism of Real-Time Maude has met all the challenges listed above, while offering simulation, and further model checking capabilities.

4.2 Ambiguities and implicit assumptions in the original specification

An informal algorithm description is rarely without implicit assumptions or ambiguities. Some implicit assumptions in the description are deliberate as they are considered trivial, but other more intricate issues demand a bit more attention. The paper [25] is rather formally written, which was highly appreciated in the modeling process. However, a few implicit assumptions, discovered in the formalization process, were not trivial. It is desirable to model the specification of the algorithm in collaboration with the developers, to achieve a specification that fully agrees with the intended interpretation of the algorithm. Making wrong assumptions could cause the specification to be incorrect. We, therefore, tried to contact one of the developers via email, to get an opinion on some of the issues, most of which resolved themselves after more careful reading. This email probably got lost somewhere, so we have to make a few assumptions on our own.

The most significant implicit assumptions that are made explicit in the Real-Time Maude model are outlined next, so that these are settled before summarizing my specification in the following sections.

Settling the assumptions

The definition of the node selection phase seems somewhat unclear to us in [25] as the definition appears circular. The node selection phase starts with the volunteering process, where each node decides probabilistically whether it wants to be a starting node. The exact time when this

should happen is unclear. We decided that this process should start at the very beginning of each round of OGDC.

It is hardly mentioned at all in [25] how a node should check whether its own coverage area is covered by other nodes. It is, however, explained how to check coverage in the entire sensing area, by dividing the area into a grid and checking if the center of each grid is covered by at least one node. In addition, it is briefly mentioned in their definition of k-coverage: "a node is only switched off when each grid point in the node's coverage area is covered by least k other nodes." This implies that a grid also should be used to check each node's coverage area. In the preliminary version of the paper, an analogous method is used to check the node's own coverage area by dividing it into a grid and use a bitmap to keep track of which sections that are covered by other nodes. This approach is used in the implementation; each node has its own bitmap to keep track of its own coverage area, as explained in Section 4.7.

I assume that all sections of a node's coverage area that are *outside* the sensing area are not taken into consideration when checking whether the node's coverage area is covered by other nodes. This is a reasonable assumption because sensing is only performed inside the sensing area. A node can therefore be switched off even if sections of its coverage area *outside* the sensing area are not covered, as long as its entire coverage area *inside* is covered by other nodes.

It is not specified in [25] how much power a node consumes when it is in the "undecided" state. Since it is assumed that the nodes do not perform any sensing tasks in this state, it is also reasonable to assume that a node consumes the same amount of power in the "undecided" state as in the "off" state.

One *power unit* is defined in [25] to be the amount of power a node consumes per second. I use a similar strategy but per millisecond. This is to make the value of each node's remaining power more readable.

For simplicity purposes Zhang and Hou abstract from the three dimensional space to the two dimensional surface in their proof in [25] that coverage implies connectivity when the transmission range is at least twice as large as the sensing range. I make the same abstraction and assume that the nodes are located on a two dimensional surface. The main part of the model that has to be changed to model the OGDC algorithm in the three dimensional space is the bitmap. The main challenge here is to give the bitmap the same intuitive format, which is not essential to faithfully model the OGDC algorithm.

The transmission time of a power-on message is implicitly assumed in [25] to be independent of the distance from the sender to the receiver. This is a reasonable abstraction since the possible differences in transmission time to the different receiving nodes are sufficiently small to be insignificant compared to the differences the receiving nodes' resulting backoff timers. The transmission time is in [25] set to 6.8 milliseconds, but is rounded off to 7 milliseconds in the model to be able to use the natural numbers as time domain as explained in Section 4.3. This is reasonable because it is said in [25] that "(OGDC's) performance is rather insensitive to the parameter values...", and the round-off error is small.

4.3 Modeling time and time elapse

As mentioned in Chapter 2, time plays a central role in the OGDC algorithm. After the assumed time synchronization protocol has been used, we need to model the advance of time in the specification. First, the *time domain* has to be defined. Real-Time Maude supports user defined time domains. However, I use a built-in time domain, `NAT-TIME-DOMAIN-WITH-INF`, which sets the time domain to be the natural numbers, extended with an infinite value `INF`. I initially used the built-in time domain `POSRAT-TIME-DOMAIN-WITH-INF` which uses rational numbers. We decided to change the time domain to the natural numbers because this time domain guarantees

that all reachable states starting from an initial state can be investigated in the analysis, and the only change done in the model is to round off the transmission time as explained in Section 4.2.

The effect and constraint of time elapse is conveniently modeled using a technique suggested by Ölveczky and Meseguer in [18, 15] for modeling time dependent behavior in object oriented specifications. They define two functions: `delta` and `mte`. The function `delta` defines the effect of time elapse on all objects and messages in the configuration, and the `mte` function defines the *maximum time elapse*, that is, the maximum amount of time that can elapse before an action has to take place. These two functions distribute over all the objects and messages in the configuration:

```
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 `delta` and `mte` functions also need to be defined for single objects and messages in the configuration as described in Sections 4.4 to 4.6.

The `delta` and `mte` functions are applied on the specification when the tick rule is used. I use the standard tick rule of Ölveczky and Meseguer 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 time `T` less than or equal to `mte(C)`. The concrete value of `T` is not defined until a *time sampling strategy* is chosen. The time sampling strategy defines the largest "step" time can advance with. Using the natural numbers as the time domain guarantees that all behaviors can be investigated by setting this "time step" to 1. The time sampling strategy is chosen in Chapter 5.

4.4 The definition of a node

Each node is identified by its geographical location, of which it is aware since we assumed that a localization protocol has been used. The sort `Location` is used as the object identifier (`Oid`) the node objects. The position of a node is measured in centimeters, which should be sufficiently accurate to keep nodes from getting the same identifier. None of the nodes got the same object identifier in any of the random generations of nodes in the analysis of the specification.

```
sort Location .
subsort Location < Oid .

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

Even though the location of a node is measured in integral centimeters, the rational numbers (`Rat`) are used to represent the sort `Location`. The reason for this is that the same sort is used to compute coverage area crossings where more accuracy is needed. An example term of the sort `Location` is `(45 . 3/2)`, meaning the location 45 centimeters along the x-axis and 3/2 centimeters along the y-axis, in a fixed coordinate system.

A node in the wireless sensor network is represented in the Real-Time Maude specification as an instance of the `WSNode` class:

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

The sorts `Status`, `VolunteeredStatus` and `NeighborList` are defined as follows:

```
sort Status .
op on        : -> Status [ctor format (g o)] .
op off       : -> Status [ctor format (r o)] .
op undecided : -> Status [ctor] .
```

The `format` attribute makes the `on` term green and the `off` term red for easier reading of the node's `status` attribute. The `format` attribute is further used to format a term of the sort `Bitmap` in Section 4.7.

```
sort VolunteeredStatus .
subsort Bool < VolunteeredStatus .
op undecided : -> VolunteeredStatus [ctor] .
```

The built-in sort `Bool` has terms `true` and `false` and the subsort declaration also makes these terms of the sort `VolunteeredStatus` in addition to the term `undecided`.

```
sort Neighbor NeighborList .
subsort Neighbor < NeighborList .
op _starting_ : Location Bool -> Neighbor .
op nil        : -> NeighborList [ctor] .
op __         : NeighborList NeighborList -> NeighborList [ctor assoc id: nil] .
```

A `Neighbor` is represented by the location of the neighbor and whether it is a starting node. A neighbor is a starting node if, and only if, the direction field in its power-on message is a non-negative number. A starting neighbor at location `(2/5 . 65)` is represented by the term `(2/5 . 65 starting true)`. Terms of the sort `Neighbor` are concatenated into a term of sort `NeighborList` by the `__` operator. The operator is declared associative (the `assoc` attribute) which means that list can be grouped in any way.

The node's attributes represent the following:

- `backoffTimer` - The node's backoff timer is a countdown for when the node should perform an action, and is set to appropriate values as outlined in Section 2.1.1.

- `bitmap` - The node's bitmap denotes what sections of the node's coverage area are covered by its neighbors (see Section 4.7).
- `neighbors` - The list of the node's neighbors, that is, the nodes it has received a power-on message from in the current round.
- `remainingPower` - The amount of power the node has left.
- `roundTimer` - The amount of time left until the next round of OGDC starts.
- `status` - The node's status, `on`, `off` or `undecided`. Previously called the node's state.
- `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.

Timed behavior for a node

The effect of time elapse on a `WSNode` object needs to be defined by an appropriate definition of the function `delta`. Each `WSNode` object has a backoff timer and a round timer which need to be updated according to the amount of time that has elapsed. Furthermore, as time passes, the node consumes different amounts of power according to what status it has:

```
var O : Oid .
var T : Time .
vars TI TI' : TimeInf .
var S : Status .
var R : Rat .
var V : VolunteeredStatus .

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

The constants `idlePower` and `sleepPower` define the amount of power the node consumes per time unit (millisecond) when has status "on" and any other status ("off" or "undecided") respectively. The *idle : sleep* power ratio is defined in [25] to 400 : 1:

```
op powerUnit : -> Nat .    eq powerUnit = 400 .
op sleepPower : -> Nat .   eq sleepPower = powerUnit / 400 .
op idlePower : -> Nat .    eq idlePower = powerUnit .
```

The node's two timers are updated by subtracting the amount of time that has elapsed with the `monus` function.

Next, we need to define the function `mte`, defining the maximum time that can elapse before a `WSNode` object must perform an instantaneous transition. Time should not be allowed to advance 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 .
```

If the node is dead it should not put any constraint on the maximum amount time is allowed to advance:

```
eq mte(< O : WSNode | remainingPower : 0 >) = INF .
```

If the node is alive, time should not advance beyond the expiration of either the backoff timer, or the round timer, to ensure that the other rules are used at the appropriate times. Additionally time should not advance beyond the time left until the node is out of power, to ensure that the node dies exactly when it is out of power (see Section 4.10.4):

```
    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 P > 0 /\ V =/= undecided .
```

4.5 Modeling communication

The method of communication is, as explained earlier, broadcasting with limited transmission range and transmission delay. The advantage of Real-Time Maude, is that it is not based on a fixed form of communication. Modeling communication is left to the programmer. Broadcast in wireless sensor networks using limited radio communication has, to our knowledge, not been previously modeled in Real-Time Maude and therefore offers some new challenges. One difficulty is that a node is not aware of its surrounding topology. That is, a broadcasting node does not know which, or how many, nodes are able to receive the broadcast.

Limited radio broadcast can elegantly be modeled as suggested by Ölveczky in private conversation. The idea is to break down the broadcast message to single, *addressed* messages to nodes that are within transmission range. This is both practical, in the sense that no unnecessary messages will be released into the configuration, and is in compliance with how broadcast logically works, since the broadcasting node will only send *one* message and the nodes within transmission range is able to receive the broadcast independently. How transmission delay is modeled is explained toward the end of this section.

In the model, a node that wants to broadcast a message does so by sending a broadcast message into the configuration. The broadcast message is then broken down into a set of addressed power-on messages. The declarations of the broadcast and power-on messages are:

```
msg broadcastFrom_withDirection_ : Oid Int -> Msg .
msg PowerOnMsgFrom_to_withDirection_ : Oid Oid Int -> Msg .
```

The broadcast message is broken down with a recursive distribution

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

The `Configuration` attribute is `frozen`, so the configuration can not be rewritten when the broadcast is being transformed, since this is logically an instantaneous transformation. Then,

by utilizing the `{_}` operator in Real-Time Maude, which encapsulates the *whole* configuration, the broadcast message is broken down to single addressed power-on messages to nodes that are within transmission range as follows:.

```
var O O' : Oid .
var D : Int .
var C : Configuration .

eq {(broadcastFrom O withDirection D) C} = {distributeMsg(O, D, C)} .
```

The variable `O` is the object identifier for the broadcasting node and `D` is the direction needed to compute the backoff timers as explained in Chapter 2. Because the sort `Configuration` is declared both associative and commutative, configuration matches the left hand side of the above equation each time there exists one or more broadcast messages in the configuration. The variable `C` contains the rest of the objects and messages in the configuration, which the distribution function traverses recursively. The distribution function has to be defined for all objects and messages in the configuration to be able to create addressed power-on messages to the nodes that are within transmission range.

Both messages and the `RandomNGen` object are released without any modifications and the distribution function continues traversing the rest of the configuration:

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

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

For each `WSNode` object in the configuration the distribution function creates an addressed power-on message to the node depending on whether the node (`O'`) is within transmission range of the broadcasting node (`O`) and continue traversing the configuration:

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

The function `withinTransmissionRangeOf` calculates the distance between the broadcasting node and the potential receiver node using Pythagoras' Theorem, and returns `true` if, and only if, the distance is less than or equal to the transmission range[1]:

```
vars O O' : Oid .
vars X X' Y Y' : Rat .

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

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

[1]We compute with the distance *squared* to avoid square roots

The power-on messages are not instantaneous. Because of the wireless capacity of radio transmission and the size of the message sent, the power-on messages take some time to reach their destinations. The technique suggested in [15] is used to model the transmission delay, by enclosing a power-on message in a *message wrapper* that adds the delay to the message:

```
op dly : Msg Time -> Msg [ctor right id: 0] .
```

The attribute `right id: 0` causes `dly(Msg, 0)` = `Msg`, and the `delta` and `mte` functions can be easily defined for messages as explained below. The transmission delay is set to 7 milliseconds:

```
op transmissionDelay : -> Nat .
eq transmissionDelay : = 7 .
```

The recursive distribution function calls are terminated when the complete set of all objects and messages have been traversed:

```
    eq distributeMsg(O, D, none) = none .
```

The effect and constraint of time elapse on power-on messages have be defined by the `delta` and `mte` functions:

```
vars T T' : Time .

eq delta(dly(PowerOnMsgFrom O to O' withDirection D, T), T') =
   dly(PowerOnMsgFrom O to O' withDirection D, T monus T') .

eq mte(dly(PowerOnMsgFrom O to O' withDirection D, T)) = T .
```

The `delta` function decreases the remaining delay by subtracting the amount of time that has elapsed. The `mte` function stops time from advancing beyond the amount of time left until the power-on message has reached its destination, so that the power-on message is received by the designated node at the correct time, as explained in Section 4.10.2. The `delta` and `mte` functions match power-on messages without the message wrapper as well, because the `dly` operator is defined with the `right id: 0` attribute. Therefore, the `mte` of a *ripe* power-on message is 0, ensuring that such messages must be treated exactly when they arrive.

4.6 Random number generation

Several parts of the OGDC algorithm are based on probabilistic behavior. The volunteering process is completely dependent on probabilities, since the process decides whether a node volunteers be a starting node. The broadcast messages from the starting nodes contain a random direction, which the computation of the optimal position is based on. Finally, the backoff timer computed upon receiving a power-on message, contains a random number. In the analysis part of the thesis, random numbers are also handy to generate different initial states, since the nodes should be *uniformly randomly* distributed in the sensing area.

I will use an object of the class `RandomNGen`, which is also used in [18], to generate random numbers. It carries a "seed" from which the random numbers is computed, with the function:

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

This random number generator should be sufficient for our use as it satisfies the criteria of a "good" random function given in [12].

The functions `delta` and `mte` must also be defined for `RandomNGen` object. This object should not change with time, and should not set a constraint on the maximum time elapse:

```
eq delta(< Random : RandomNGen | >, T') = < Random : RandomNGen | > .
eq mte(< Random : RandomNGen | >) = INF .
```

4.7 The node's coverage area

A crucial part of OGDC is checking whether a node's coverage area is completely covered by other nodes, since this is what solely[2] decides whether a node can be switched off. The bitmap consists of the following sort and subsort declarations:

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

Each node's coverage area is divided into a *grid*, and each bit in the bitmap represents the *center* of a grid square and has one of three values: `t` if the location of the bit is covered by at least one node, `f` if the location is not covered or `'`:

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

The reason for the bogus `'` bit is that the bitmap is square the coverage area is circular so it is used to "pad" the circles (see Figure 4.1). This bit is also used to denote the sections of a node's coverage area which are outside the entire sensing area.

A `Bitmap` consists of several `BitList`s of `Bit`s. The `Bit`s are concatenated into a `BitList`:

```
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. Each `BitList` is enclosed by `|..|`:

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

Here, a new line is inserted before each `BitList` by using the `format` attribute. The `BitLists` are then concatenated into a `Bitmap`:

```
op nil : -> Bitmap [ctor] .
op __ : Bitmap Bitmap -> Bitmap [ctor assoc id: nil ] .
```

The Maude tool is by no means a graphical tool, but with proper use of the `format` attribute, the bitmap is given an intuitive appearance as shown in Figure 4.1.

[2]When it is assumed that the radio range is at least twice as large as the sensing range.

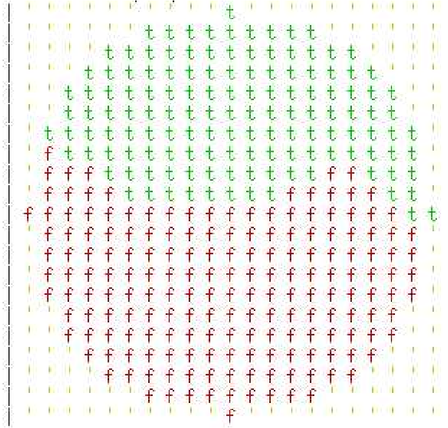


Figure 4.1: The bitmap for a node's coverage area.

4.7.1 Initializing the bitmap

Only the value of each bit in the bitmap is "stored" in the bitmap. The location of each bit is computed from the location of the node since this is the center of the bitmap. The distance between the bits are defined by the constant `bitInc`, from which the number of bits in the bitmap (`bitCounter`) is computed:

```
op bitCounter  : -> Nat .
eq bitCounter = ceiling((2 * sensingRange) / bitInc)
                + if (bitInc divides (2 * sensingRange))
                  then 1 else 0 fi .
```

The `bitCounter` denotes the size of the bitmap by the number bits the bitmap consists of in the horizontal (and vertical, since bitmap is square) direction.

A node's bitmap is created by a function `initBitmap` that has the location of the node as its only argument:

```
op initBitmap    : Location -> Bitmap .
op makeBitmap    : Oid Location Nat Nat -> Bitmap .
op makeBitList   : Oid Location Nat Nat -> BitList .
```

The function `makeBitmap` traverses the bitmap and calls `makeBitList` for each `BitList`:

```
var O : Oid .
vars X Y : Rat .
vars I N : Nat .

eq initBitmap(X . Y) =
   makeBitmap(X . Y, X - sensingRange . Y + sensingRange, bitInc, bitCounter) .

eq makeBitmap(O, X . Y, I, N) =
   if N > 0
   then | makeBitList(O, X . Y, I, bitCounter) |
        makeBitmap(O, X . Y - I, I, N - 1)
   else nil
   fi .
```

The location of the top left hand corner of the bitmap is `(X - sensingRange . Y + sensingRange)` and the location of the first bit in each `BitList` is found, starting at the first bit in the top `BitList`, by subtracting `bitInc` (the variable `I` in the `makeBitmap` equation) from the current vertical position to reach the bit below. The `bitCounter` (`N`) argument is decremented for each recursive function call, and the recursion is terminated when the argument reaches `0`. The function `makeBitList` creates the actual bits in the bitmap and each bit's value is dependent on whether the bit is within the sensing range of the node and if it is within the sensing area:

```
eq makeBitList(O, X . Y, I, N) =
   if N > 0
   then (if (O withinSensingRangeOf (X . Y)) and isInSensingArea(X
         then f
         else '
         fi)
         makeBitList(O, X + I . Y, I, N - 1)
   else nil
   fi .
```

The bits in the bitmap are created by recursive function calls on `makeBitList` and the next bit in the `BitList` is created by adding the value of the variable `I` (`bitInc`) to the current horizontal position. The recursion is terminated when the argument `N` (`bitCounter`) reaches 0. All the bits in the bitmap that are within the sensing range of the node and within the sensing area are initialized as uncovered by the `f` bit. Otherwise it is initialized to `'`.

4.7.2 Updating the bitmap

A node's bitmap is updated each time the node receives a power-on message, according to the overlap of its coverage area and the coverage area of sender of the power-on message, called the "new neighbor".

```
op updateBitmap : Oid Bitmap Oid -> Bitmap .
op updateBitmap : Bitmap Oid Location Nat -> Bitmap .
op updateBitList : BitList Oid Location Nat -> BitList .
```

The function `updateBitmap` changes the bits in the receiving node's (first `Oid` in the declaration) bitmap that are within the new neighbor's (second `Oid`) coverage area to `t`, if the bit is not already set to this value. The function traverses the bitmap in the same manner as the `initBitmap` function. It calls a recursive function, also called `updateBitmap`, which again calls the `updateBitList` function for each `BitList` in bitmap:

```
var L : Location .
vars BM  : Bitmap .
var BITL : BitList .
var BIT : Bit .

eq updateBitmap(X . Y, BM, O) =
   updateBitmap(BM, O, (X - sensingRange . Y + sensingRange), bitInc) .

eq updateBitmap(nil, O, L, I) = nil .

eq updateBitmap(| BITL | BM, O, X . Y, I) =
```

```
     | updateBitList(BITL, O, X . Y, I) |
 updateBitmap(BM, O, X . Y - I, I) .
```

The `updateBitList` function traverses the bits in the `BitList` recursively. It keeps the bit intact if it has the `t` or `'` value, otherwise it updates the bit if its location is within the new neighbors coverage area:

```
eq updateBitList(nil, O, L, I) = nil .
eq updateBitList(BIT BITL, O, X . Y, I) =
    if (BIT =/= f)
    then BIT
    else if O withinSensingRangeOf (X . Y)
         then t
         else f
         fi
    fi
    updateBitList(BITL, O, X + I . Y, I) .
```

The node's coverage area is considered covered when all the bits in the bitmap is set to either `t` or `'`. This is done by simply traversing the bitmap and checking the value of each bit with a function `coverageAreaCovered` that returns a boolean value:

```
eq coverageAreaCovered(nil) = true .
eq coverageAreaCovered(| nil | BM) = coverageAreaCovered(BM) .
eq coverageAreaCovered(| BIT BITL | BM) =
   (BIT =/= f) and coverageAreaCovered(| BITL | BM) .
```

4.8 Computing the coverage area crossings

The computations required of the crossings between two nodes' coverage areas was found in a preliminary version of [25]. A crossing is represented in the model by the two nodes that create the crossing, and the location of the crossing:

```
op _x_in_ : Oid Oid Location -> Crossing [ctor] .
```

An example term of sort `Crossing` where nodes `(10 . 0)` and `(20 . 0)` creates a crossing at location `(15 . 5) is:`

```
(10 . 0) x (20 . 0) in (15 . 5)
```

A list of crossings is defined in the standard way:

```
op nil : -> CrossingList [ctor] .
op __ : CrossingList CrossingList -> CrossingList [ctor assoc id: nil] .
```

When a node receives a power-on message, it checks if it has uncovered crossings within its coverage area. This is done by extracting the locations of the node's neighbors from the list of neighbors with a function `extractOid`, and first compute *all* the crossings created by its neighbors that are connected:

```
op findAllCrossings : OidSet -> CrossingList .

vars O O' : Oid .
```

```
var OS    : OidSet .

eq findAllCrossings(none) = nil .
eq findAllCrossings(O) = nil .
eq findAllCrossings(O ; O' ; OS) = findAllPairCrossings(O ; OS)
                                   findAllCrossings(O' ; OS) .
```

All the computations needed to find the location of a crossing are done by the function `findAllPairCrossings` which returns a list of all crossings created by the set of neighbors by traversing this set and computes the crossings between each pair. The crossings that are outside its coverage area are removed.

```
op extractValidCrossings : Oid CrossingList -> CrossingList .

var O'' : Oid .
var L   : Location .
var CL  : CrossingList .

eq extractValidCrossings(O, nil) = nil .
eq extractValidCrossings(O, (O' x O'' in L) CL) = if O withinSensingRangeOf L
                                                  then (O' x O'' in L)
                                                  else nil
                                                  fi
                                                  extractValidCrossings(O, CL) .
```

Then only the crossings that are not within the coverage area of a third neighbor are kept.

```
op listUncoveredCrossings : OidSet CrossingList -> CrossingList .

eq listUncoveredCrossings(OS, nil) = nil .
eq listUncoveredCrossings(OS, (O x O' in L) CL)
  =
   if checkSingleCrossing(OS minus O ; O', (O x O' in L))
   then (O x O' in L)
   else nil
   fi
   listUncoveredCrossings(OS, CL) .
```

All the uncovered crossings within the coverage area of a node `O` with neighbors `NBL` are then given by:

```
listUncoveredCrossings(extractOid(NBL),
                       extractValidCrossings(O, findAllCrossings(extractOid(NBL))))
```

4.9 Computing the backoff timers

The backoff timers that need to be computed are T_a and T_b from Section 2.1.4, which are computed when a node receives a power-on message. They are implemented by the functions `setTa` and `setTb` and are computed from the formulas given in [25]:

```
op setTC1 : Oid NeighbourList Nat -> Time .
op setTC2 : Oid Oid Nat NeighbourList Nat -> Time .
```

```
vars O O' : Oid .
var NBL : NeighborList .
vars N D : Nat .

eq setTa(O, NBL, N) = ceiling(t0 * (c *
    (((sensingRange - dTa(O, NBL)) * (sensingRange - dTa(O, NBL)))
    + (dTa(O, NBL) * dTa(O, NBL) * dAlphaTa(O, NBL) * dAlphaTa(O, NBL)))
   + randomU(N))) .


eq setTb(O, O', D, NBL, N) = ceiling(t0 * (c *
    (((sqrt(3) * sensingRange - dTb(O, O')) * (sqrt(3) * sensingRange - dTb(O, O')))
    + (dTb(O, O') * dTb(O, O') * dAlphaTb(O, O', D) * dAlphaTb(O, O', D)))
   + randomU(N))) .
```

The function `dTa` computes the distance between the receiving node `O` and the closest crossing created by the neighbor list `NBL` (see Figure 4.2(a)), and `dTb` computes the distance between the receiving node and the broadcasting node `O'` (see Figure 4.2(b)):

```
op dTC1 : Oid NeighbourList -> Rat .
op dTC2 : Oid Oid -> Rat .
```

The function `dAlphaTa` computes the angle between the optimal position with respect to closest uncovered crossing and the receiving node[3] (see Figure 4.2(c)), and `dAlphaTb` computes the angle between the optimal position with respect to a single starting node and the receiving node (see Figure 4.2(d)):

```
op dAlphaTC1 : Location NeighbourList -> Rat .
op dAlphaTC2 : Location Location Nat -> Rat .
```

The definitions of these functions are in Appendix A.

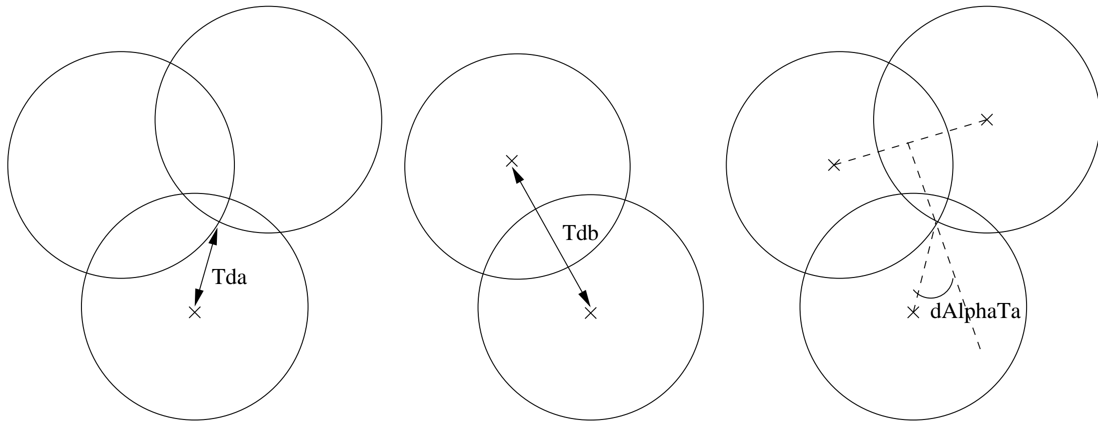
4.10 The OGDC algorithm

The dynamic behavior of the Real-Time Maude model is modeled by the rewrite rules. When the specification is modeled on a suitable level of abstraction, the model acts as a formal algorithm description as well as being a completely executable specification, which can be subjected to simulations and thorough analysis.

The rules are grouped into those that relate to the volunteering process (Section 4.10.1), rules that handle power-on messages (Section 4.10.2), and rules that take care of a node becoming active (Section 4.10.3). Section 4.10.4 contains the rules that specify the actions taken when a node dies and when a new round begins. The commonly used variables used in the rules are:

```
var B : Bool .
vars O O' : Oid .
var D : Int .
vars M N : Nat .
vars P R : Rat .
```

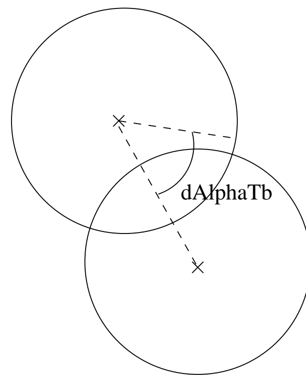
[3]Here lies a bug that was not fixed by the completion of this thesis. If the receiving node is in the exact position of the uncovered crossing, the angle can not be computed



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

(b) The distance between the receiving node and the broadcasting node.

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



(d) The angle between the optimal position with respect to a single starting node and the receiving node.

Figure 4.2: Distances and angles in the computation of backoff timers.

```
var L : Location .
var BM : Bitmap .
vars NB NEW_NEIGHBOR : Neighbor .
var NBL : NeighborList .
var T : Time .
var S : Status .
```

4.10.1 The volunteering process

Each node enters its volunteering process at the beginning of each round, which kicks off the node selection phase, and is modeled by `volunteer` rule. The definition of `mte` (see Section 4.4) stops time from advancing until each node's `hasVolunteered` attribute is changed to either `true` or `false`, and the only rule that can be applied is the `volunteer` rule. Since all node objects are initialized with `hasVolunteered` attribute set to `undecided`, this forces the volunteering process to be started at the beginning of each round:

```
crl [volunteer] :
   < O : WSNode | remainingPower : P,
                  volunteerProb : R,
                  hasVolunteered : undecided >
   < Random : RandomNGen | seed : M >
 =>
   (if (randomProb(M) < R) and (P > powerThreshold or R == 1)
    then < O : WSNode | backoffTimer : randomTimer(random(M)),
                        hasVolunteered : true >
    else < O : WSNode | backoffTimer : nonVolunteerTimer,
                        hasVolunteered : false >
    fi)
    < Random : RandomNGen | seed : repeatRandom(M, 3) >
 if P > 0 .
```

The `RandomNGen` object `Random` carries the seed which is used to compute random numbers. The seed is updated with a new seed according to the number of random numbers that has been computed in the rule.

Whether the node volunteers to be a starting node depends on probability (`randomProb(M) < N`), and the node also either has to have sufficient remaining power (`P > powerThreshold`) or its volunteer probability must have reached 1 (`N == 1`). The `randomProb` function generates a random number from the seed the `RandomNGen` object between 0 and 100:

```
op randomProb : Nat -> Nat .
eq randomProb(N) = random(N) rem 1000 .
```

The constant `powerThreshold` is initialized according to [25], to the amount of power required to stay idle for 900 seconds:

```
op powerThreshold : -> Nat .
eq powerThreshold = 900000 * powerUnit .
```

If the node volunteers, it sets its backoff timer to a random value between 0 and 10:

```
op volunteerTimeBound : -> Time .
eq volunteerTimeBound = 10 .

eq randomTimer(N) = random(N) rem volunteerTimeBound .
```

When a volunteered node's backoff timer expires it becomes an active node, as explained in Section 4.10.3. If the node does not volunteer, it sets its backoff timer to a constant `nonVolunteerTimer`:

```
op nonVolunteerTimer : -> Time .
eq nonVolunteerTimer = 1000 .
```

When a non-volunteered node's backoff timer expires without having received a single power-on message (`neighbors : nil`), it repeats the volunteering process with the probability for volunteering doubled:

```
crl [repeatVolunteering] :
    < O : WSNode | backoffTimer : 0, neighbors : nil,
                   remainingPower : P, volunteerProb : R,
                   hasVolunteered : false >
  =>
    < O : WSNode | backoffTimer : INF, volunteerProb : doubleProb(R),
                   hasVolunteered : undecided >
 if P > 0 .
```

Remember that time will not elapse when `hasVolunteered` is `undecided`, so the `volunteer` rule must be applied immediately.

4.10.2 Ignoring and receiving power-on messages

To model transmission delay, the power-on message is enclosed within the `dly` operator when broadcast, as explained in Section 4.5. A node is not allowed to receive the power-on message until it has reached the node, that is, when the delay of the power-on message has expired. The power-on message only appears "free" of message wrapper when its delay reaches 0 (`dly(`$power-onMsg$`,0) =` $power-onMsg$). The power-on message is then ripe, and can be received by the designated node. Each rule in this section therefore can only be applied when a ripe power-on message and its destination node exists in the configuration.

A power-on message is ignored either when the receiving node is already active:

```
crl [ignoreMsg] :
    (PowerOnMsgFrom O' to O withDirection D)
    < O : WSNode | remainingPower : P, status : on >
  =>
    < O : WSNode | >
 if P > 0 .
```

or if the node is dead:

```
eq (PowerOnMsgFrom O' to O withDirection D)
   < O : WSNode | status : off, remainingPower : 0 >
  =
   < O : WSNode | > .
```

Ignoring a power-on message when the node is dead is modeled as an equation instead of a rule. The reason for this is that, logically, a node that is dead does not actively make a decision to ignore the power-on message. It can simply not receive the message because it is dead. Furthermore, by using an equation instead of a rule reduces the state space in the specification, which is practical in the formal analysis of the specification.

When the new neighbor causes the receiving node's coverage area to be covered by its neighbors (`coverageAreaCovered(..)`), it can be switched off:

```
crl [recPowerOnMsgAndSwichOff] :
    (PowerOnMsgFrom O' to O withDirection D)
    < O : WSNode | status : S, neighbors : NBL,
                   bitmap : BM, remainingPower : P >
  =>
    < O : WSNode | status : off, neighbors : NBL NEW_NEIGHBOR,
                   bitmap : updateBitmap(O, BM, O'),
                   backoffTimer : INF >
  if NEW_NEIGHBOR := (O' starting (D >= 0)) /\
     S =/= on /\ P > 0 /\
     coverageAreaCovered(O, updateBitmap(O, BM, O'), O') .
```

The variable `NEW_NEIGHBOR` is assigned the value `(O' starting (D >= 0))` which represents the sender of the power-on message (`O'`) as a starting or non-starting neighbor depending on whether the value of the direction in the power-on message is non-negative.

The next three rules correspond to the three conditions a, b and c listed in Section 2.1.4. These rules can only be applied when the new neighbor does not cause the receiving node's coverage area to be covered (`not coverageAreaCovered(...)`), and all three rules add the sender of the power-on message to the receiving node's list of neighbors and updates the receiving node's bitmap.

If a node receives a power-on message and has at least one uncovered crossing within its coverage area (`existsUncoveredCrossings(...)`), it (re)sets its backoff timer to T_a (`setTa(...)`) if the sender of the latest power-on message creates the *closest* uncovered crossing (`O' createsClosestCrossingTo O withNeighbors (NBL NEW_NEIGHBOR)`):

```
crl [recPowerOnMsgWithUncoveredCrossings] :
    (PowerOnMsgFrom O' to O withDirection D)
    < O : WSNode | remainingPower : P, status : S,
                   backoffTimer : T,
                   neighbors : NBL, bitmap : BM >
    < Random : RandomNGen | seed : M >
  =>
    < O : WSNode | backoffTimer : (if O' createsClosestCrossingTo
                                   O withNeighbors (NBL NEW_NEIGHBOR)
                                   then setTa(O, NBL NEW_NEIGHBOR, M)
                                   else T
                                   fi),
                   neighbors : NBL NEW_NEIGHBOR,
                   bitmap : updateBitmap(O, BM, O') >
    < Random : RandomNGen | seed : random(M) >
 if NEW_NEIGHBOR := (O' starting (D >= 0)) /\
    S =/= on /\ P > 0 /\
    not coverageAreaCovered(O, updateBitmap(O, BM, O'), O') /\
    existsUncoveredCrossings(O, NBL NEW_NEIGHBOR) .
```

If a node does not have an uncovered crossing within its coverage area but at least one of its neighbors is a starting node (`existsStartingNeighbor(NBL NEW_NEIGHBOR)`), it (re)sets its

backoff timer to T_b (`setTb(...)`) if the sender of the latest power-on message is the *closest* starting neighbor (`O' == findClosestStartingNeighbor(O, NBL NEW_NEIGHBOR`):

```
crl [recPowerOnMsgWithStartingNeighbors] :
    (PowerOnMsgFrom O' to O withDirection D)
    < O : WSNode | remainingPower : P, status : S, backoffTimer : T,
                   neighbors : NBL, bitmap : BM >
    < Random : RandomNGen | seed : M >
  =>
    < O : WSNode | backoffTimer : (if O' == findClosestStartingNeighbor(O,
                                              NBL NEW_NEIGHBOR)
                                   then setTb(O, O', D,
                                        NBL NEW_NEIGHBOR, M)
                                   else T
                                   fi),
                   neighbors : NBL NEW_NEIGHBOR,
                   bitmap : updateBitmap(O, BM, O') >
    < Random : RandomNGen | seed : random(M) >
 if NEW_NEIGHBOR := (O' starting (D >= 0)) /\
    S =/= on /\ P > 0 /\
    existsStartingNeighbor(NBL NEW_NEIGHBOR) /\
    not coverageAreaCovered(O, updateBitmap(O, BM, O'), O') /\
    not existsUncoveredCrossings(O, NBL NEW_NEIGHBOR) .
```

If a node does not have an uncovered crossing within its coverage area and none of its neighbors are starting nodes, it (re)sets its backoff timer to T_c (`tc`) if the sender of the latest power-on message is the *closest* neighbor (`O' == findClosestNeighbor(O, NBL NEW_NEIGHBOR)`):

```
crl [recPowerOnMsgWithNeighbors] :
    (PowerOnMsgFrom O' to O withDirection D)
    < O : WSNode | remainingPower : P, status : S, backoffTimer : T,
                   neighbors : NBL, bitmap : BM >
  =>
    < O : WSNode | backoffTimer : (if O' == findClosestNeighbor(O,
                                              NBL NEW_NEIGHBOR)
                                   then tc
                                   else T
                                   fi),
                   neighbors : NBL NEW_NEIGHBOR,
                   bitmap : updateBitmap(O, BM, O') >
  if NEW_NEIGHBOR := (O' starting (D >= 0)) /\
     S =/= on /\ P > 0 /\
     not existsStartingNeighbor(NBL NEW_NEIGHBOR) /\
     not coverageAreaCovered(O, updateBitmap(O, BM, O'), O') /\
     not existsUncoveredCrossings(O, NBL NEW_NEIGHBOR) .
```

4.10.3 The activation of a node

A node becomes active either when a volunteered node's backoff timer expires, or upon the expiration of the backoff timer of a non-volunteered node that has received at least one power-on message.

When a volunteered node's backoff timer expires, it becomes active as a starting node and broadcasts a power-on message that contains the node's location and a *random direction*.

```
crl [startingNodeSwitchOn] :
    < 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) >
    broadcastFrom O withDirection randomDirection(M)
 if P > 0 .
```

The broadcast consumes `transPower` amount of power where the *idle : transmit* power ratio is defined in [25] to 1 : 5. That is, the node consumes 5 times as much power per millisecond when transmitting a message as it does when idling:

```
op transPower : -> Nat .   eq transPower = powerUnit * 5 * transmissionDelay .
```

If a non-volunteered node's backoff timer expires after it has received at least one power-on message (`neighbors : NB NBL`), it becomes an active node as a non-starting node and broadcasts a power-on message that contains the node's location and *-1* in the direction field:

```
crl [nonStartingNodeSwitchOn] :
    < O : WSNode | remainingPower : P, backoffTimer : 0,
                   neighbors : NB NBL, hasVolunteered : false >
  =>
    < O : WSNode | remainingPower : P monus transPower,
                   backoffTimer : INF,
                   status : on >
    broadcastFrom O withDirection -1
 if P > 0 .
```

When a node switches on it sets its `backoffTimer` attribute to `INF`. That way the node's backoff timer no longer puts a constraint on the amount of time that is allowed to elapse.

4.10.4 Death of a node and restarting the round

The death of a node is modeled as an equation. This together with the `mte` definition that stops time when a node is out of power, secures an instant death for "powerless" nodes, because a node is reduced by the equations after each rule application. So each time tick rule results in a node consuming all of its power is instantly killed by the following equation before any other rules can be applied:

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

When a node dies its `backoffTimer` and `roundTimer` attributes are set to `INF`. That way, the node does not put a constraint on the amount of time that allowed to elapse after it is dead. In case the node dies in its volunteering process, and the `hasVolunteered` attribute is `undecided`, the attribute is changed so that it does not put constraint on the amount of time that can advance (see Section 4.4). To prevent uncontrolled use of this equation, it can not be applied when the node is already dead, that is, the equation can only be applied when the node's `status` attribute is different from `off`.

When the round is over, each "living" node (`P > 0`) resets all of its attributes except the remaining power, to the initial values:

```
crl [restart] :
    < O : WSNode | roundTimer : 0, remainingPower : P >
  =>
    < O : WSNode | status : undecided,
                   neighbours : nil,
                   bitmap : initBitmap(O),
                   hasVolunteered : undecided,
                   backoffTimer : INF,
                   roundTimer : roundTime,
                   volunteerProb : 1000 / n >
 if P > 0 .
```

The constant `n` denotes the number of nodes that are in the network. It is declared in the model, but not defined:

```
op n : -> Nat .
```

This constant must be defined before analysis can be performed, and is defined according to the number of nodes in the initial state (see Section 5.1.2).

Chapter 5

Analysis of the OGDC Algorithm

This chapter illustrates some of the Real-Time Maude analysis of the OGDC algorithm.

A wireless sensor network possibly consists of thousands of nodes, which poses a significant challenge to real-time analysis tools. In [25], Zhang and Hou use a *discrete event simulator* tool called *The Network Simulator* [14] (ns-2), with the wireless extension developed by the CMU Monarch group [8], to simulate the OGDC algorithm. ns-2 is a specialized tool, developed with support from DARPA, for network simulations of, among others, TCP, routing, and multicast protocols in both wired and wireless networks. The paper [25] is first and foremost written to present the OGDC algorithm and compare it to other density control algorithms, not to illustrate the formalism or the analysis capabilities of ns-2. Zhang and Hou have performed several simulations of the OGDC algorithm to measure some essential *performance metrics*. The performance metrics are:

- The percentage of coverage provided by the active nodes.
- How percentage of coverage and total remaining power changes over time.
- The α-lifetime of the network, that is, the total time during which at least α percent of the sensing area is covered.

The results of the simulations are then used to compare the OGDC algorithm to other density control algorithms. The simulations investigate single behaviors of the OGDC algorithm. Zhang and Hou execute simulations repeatedly to provide more data, and thereby gain confidence in the results of the simulations.

Odd behaviors in a system might not be discovered during extensive simulations, as they only investigate *one* behavior. In addition to being able to execute single behaviors in a system, Real-Time Maude is able to investigate *all* behaviors from a given initial state. It is illustrated in the next sections how the simulations in [25] can be performed in Real-Time Maude, and how Real-Time Maude's other analysis capabilities can be used to perform a more thorough analysis of the OGDC algorithm in search for errors.

In my analysis, it was quickly brought to my attention that simulations, which execute one behavior in the model starting from initial states, with a large number of nodes (over 300), took an unexpectedly large amount of time to terminate. This chapter focuses on the analysis capabilities of Real-Time Maude and not on the exact reasons for the long execution times.

When the number of nodes in the network increases, the reachable state space also increases exponentially. The reason for this is the combinatorial explosion resulting from the fact that any set of nodes can volunteer to be starting nodes. Timed searches and temporal logic model checking investigates the entire state space, and are, therefore, performed with few nodes. State space explorative analysis of other communication protocols with this amount of nodes has

previously been used to find subtle but significant design errors, see e.g., [18, 9], that were not discovered using extensive simulations. This kind of analysis is, therefore, meaningful in search for errors.

Section 5.2 shows how the analysis in [25] with the ns-2 tool can be done in Real-Time Maude. Section 5.3 contains examples of how timed search and temporal logic model checking can be used to perform other types of analysis. First, in Section 5.1, the environment in which the analysis can take place is outlined.

5.1 The analysis environment

To perform the analysis, some auxiliary functions and temporal logic propositions need to be defined. These are placed in a separate module `OGDC-ANALYSIS` that includes the model of the OGDC algorithm.

5.1.1 The sensing area

All the nodes in the network are located within the sensing area. Both the size of the area and the number of nodes in the network varies when different analysis are performed. The size of the sensing area is stored as the constant `sensingAreaSize` in the `OGDC-ANALYSIS` module. The reason I need this constant is to check sensing area coverage. Coverage in the sensing area is checked, as in [25], by dividing it into a grid, where one grid square is $1m^2$, and checking which grid square *centers* are covered by at least one node. The area is implemented using the same bitmap construction as the nodes' coverage areas. The sensing area bitmap acts as a coordinate system with the origin in the center. A $50m \times 50m$ sensing area, for example, contains nodes at locations from `(-2500 . -2500)` to `(2500 . 2500)`. The sensing area bitmap is depending on the size of the sensing area, which is why this size is a constant in the `OGDC-ANALYSIS` module.

5.1.2 Generating initial states

Generating different initial states is particularly important in simulations. Here, only one behavior of the model is investigated from each initial state. It is therefore important to run simulations from several initial states, to better validate a system.

Because of the computations of the volunteer probability (see Section 4.10.4) when a new round begins, the constant `n` has to be defined when running more rounds of the OGDC algorithm. It is defined in the `OGDC-ANALYSIS` module to be the number of nodes in the initial state.

I use a function `genInitConf` to generate initial states. A term `genInitConf(M, N)` generates a `RandomNGen` object and `N` nodes at "random" locations within the sensing area:

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

The `WSNode` objects are initialized as follows:

```
< L : WSNode | remainingPower : lifetime,
               status : undecided,
               neighbors : nil,
               bitmap : initBitmap(L),
               backoffTimer : INF,
               roundTimer : roundTime,
               volunteerProb : ceiling(100 / n),
               hasVolunteered : undecided >
```

where `L` is the following randomly generated location within sensing area:

```
(random(seed) rem sensingAreaSize - (sensingAreaSize / 2)) .
(random(random(seed)) rem sensingAreaSize - (sensingAreaSize / 2))
```

The *seed* is updated for each node that is generated. The final seed, after the nodes have been generated, is set as the initial seed for the `RandomNGen` object. The constant `lifetime` denotes the node's starting power and is defined to the amount of power needed to stay idle for 5 000 000 milliseconds. The `roundTime` constant is the length of one round of OGDC, that is, 1 000 000 milliseconds:

```
eq lifetime = 5000000 * powerUnit .
eq roundTime = 1000000 .
```

The `powerUnit` constant is defined in Section 4.4. The bitmap is initialized as explained in Section 4.7.

The most natural approach would be to include the size of the sensing area as an argument to the `genInitConf` function. However, the size of the sensing area is needed for coverage measurements in other parts of the analysis as well. The sensing area size is therefore stored in the analysis module as explained in the beginning of this chapter.

We can generate an arbitrary amount of nodes located at random positions within the chosen sensing area by varying `M` and `N` in `genInitConf`. Different initial states can be created with, for instance, 100 nodes by varying the *seed*:

```
{genInitConf(seed, 100)}
```

An example of an initial state with 100 nodes is:

```
Maude> (red {genInitConf(1,100)} .)

{< Random : RandomNGen | seed : 2138 >

< 3 . 585 : WSNode |
        backoffTimer : INF,
        bitmap :(
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' ' f f f f f f f f f f f f f ' ' ' ' |
        | ' ' ' ' ' ' f f f f f f f f f ' ' ' ' ' ' |
```

```
        | ' ' ' ' ' ' ' ' ' ' f ' ' ' ' ' ' ' ' ' ' |),
        neighbors : nil,
        remainingPower : 2000000000,
        roundTimer : 1000000,
        status : undecided,
        volunteerProb : 1,
        hasVolunteered : undecided >

< 7 . 435 : WSNode |
        backoffTimer : INF,
        bitmap :(
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' ' f f f f f f f f f f f f f ' ' ' ' |
        | ' ' ' ' ' ' f f f f f f f f f ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' f ' ' ' ' ' ' ' ' ' ' |),
        neighbors : nil,
        remainingPower : 2000000000,
        roundTimer : 1000000,
        status : undecided,
        volunteerProb : 1,
        hasVolunteered : undecided >

< 13 . 151 : WSNode |
        backoffTimer : INF,
        bitmap :(
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
        | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
```

```
    | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
    | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
    | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
    | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
    | ' ' ' f f f f f f f f f f f f f f f ' ' ' |
    | ' ' ' ' f f f f f f f f f f f f f ' ' ' ' |
    | ' ' ' ' ' ' f f f f f f f f f ' ' ' ' ' ' |
    | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |),
    neighbors : nil,
    remainingPower : 2000000000,
    roundTimer : 1000000,
    status : undecided,
    volunteerProb : 1,
    hasVolunteered : undecided >

[...] }
```

I use the notation `[...]` to denote that output is removed. The reason why the coverage areas of these three nodes are not circular is that parts of their coverage areas are outside the sensing area. The bits in the coverage are that is outside the sensing area are treated as though they are covered, and are initialized to `'`.

5.1.3 The time sampling strategy

As mentioned in Section 4.3, a time sampling strategy must be chosen before the analysis can take place. This defines which points in time that are visited. All analysis is, therefore, done with respect to the chosen time sampling strategy. The time domain is the natural numbers, which guarantees that all behaviors starting from the initial state can be investigated by visiting *all* points in time, that is, by setting the time sampling strategy to increase time in steps of 1:

```
set tick def 1 .
```

However, all events in the OGDC algorithm happen at a specific time, triggered either by the expiration of timers or by the arrival of power-on messages. Therefore, the only points in time we need to visit are when these events must take place. We do not loose any behaviors in the system by "fast forwarding" between the events. If the next event happens in 10 time units, advancing time in steps of less than 10 does *not* result in any new behaviors, because no rules can be applied at these times. The next event in the system is defined by the `mte` function. Time can, therefore, advance by an arbitrary amount, as long as it is less than or equal to `mte(C)`. So, for efficiency purposes, we use a time sampling strategy that "fasts forward" to the point in time when the next event must happen:

```
set tick max def roundTime .
```

This time sampling strategy 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). This is the time sampling strategy used in the analysis, unless another is specified.

5.2 The ns-2 simulations of the OGDC algorithm in Real-Time Maude

This section explains how the analysis in [25] with the simulation tool ns-2 can be performed in Real-Time Maude. The simulations performed with the ns-2 tool provide the information

needed to decide the performance metrics listed in the beginning of this chapter. The following concrete metrics are measured:

- The number of active nodes and percentage of coverage for different numbers of deployed nodes (Section 5.2.1).
- The percentage of coverage and total amount of remaining power at different points in the networks lifetime. (Section 5.2.2).
- The α-*lifetime* for different values of α, and α-lifetime for different number of deployed nodes (Section 5.2.3).

Each simulation in [25] is repeated 20 times, an average is computed, and the results are presented in graphs.

Simulation in Real-Time Maude is done with the timed fair rewrite command `tfrew` which executes *one* behavior in the system, starting from a given initial state. By generating different initial states, as outlined in Section 5.1.2, we can easily execute more simulations. In my analysis, the performance metrics are computed *during* the simulations, and the results are found in the configuration as messages.

The general strategy I use in Real-Time Maude to compute the different metrics is to place an *analysis message*, in the initial state. This is done with a bit of abuse of the concept of messages, since the analysis messages are not sent or received by any nodes. It is simply a convenient way of computing different metrics at specific times and "storing" the result in the configuration. These messages can possibly increase the state space, but that does not affect the state space exploration, because these messages are only used in the simulations.

Each analysis message is enclosed in the `dly` operator when it is placed in the initial state. The delay denotes at what time the metric of interest should be computed. Each time an analysis message computes a metric, it duplicates itself by producing a *result message* which contains the result of the computation. These messages stay in the configuration throughout the rest of the execution so that the results can be reviewed afterwards. To achieve this, the messages are equipped with an `INF` delay. The declaration of `dly` operator must be extended to handle the `INF` value:

```
op dly : Msg TimeInf -> Msg [ctor right id: 0] .
```

The function `mte` must be defined for these new messages. The function is generalized to handle all terms of the sort `Msg`:

```
var TI : TimeInf .
var M : Msg .
eq mte(dly(M, TI)) = TI .
```

These modifications do not affect the model in anyway as they are merely generalizations. At the end of the simulation, the configuration will contain a set of result messages that can be reviewed.

The analysis messages used to compute the number of active nodes, percentage of coverage and the total remaining power are:

```
msgs activeNodes coveragePercentage totalRemainingPower : Nat Nat -> Msg .
```

When any of these messages exists in the configuration with a non-`INF` delay it is an analysis message and will compute the metric when the delay expires, that is, when the message is ripe. If the delay is set to `INF`, the message is a result message. In that case, the first argument is the

computation number (which in my simulations denotes which *round* the metric was computed), which identifies the result message, and the second contains the actual result. For example, the result message `dly(activeNodes(12, 45), INF)` denotes that the 12th computation of this metric resulted in 45 active nodes.

Each analysis message in the following sections is defined to compute the metric of interest once in every round. The delay determines the exact time the metric of interest is to be computed, and is set to `roundTime - 1` in my simulations. The reason for this is that some of the metrics need to be computed when the network is in the steady state phase. They are therefore computed at the end of each round to be completely sure that the steady state phase has been reached. The first argument in the result messages therefore denotes which *round* the computation was done, because the analysis message spawns *one* result message every round.

The next sections show how the simulations performed using ns-2 in [25] can be done with Real-Time Maude using the analysis and result messages.

5.2.1 The number of active nodes and percentage of coverage with respect to the number of deployed nodes

The simulations in [25] investigate how the number of active nodes and the percentage of coverage changes with the number of deployed nodes. This is performed in Real-Time Maude by varying the number of deployed nodes with the `genInitConf` function, and measuring the number of active nodes and percentage of coverage with the analysis messages.

Computing the number of active nodes

The number of active nodes is calculated when the analysis message `activeNodes` is ripe:

```
eq {activeNodes(N, 0) C} =
   {dly(activeNodes(N, numActiveNodes(C)),INF)
    dly(activeNodes(N + 1, 0), roundTime) C} .
```

The function `numActivNodes` computes the number of active nodes and the result is placed in the result message, which has its delay set to `INF`, so that it can stay in the configuration indefinitely. At the same time, the analysis message resets its delay to `roundTime`, so that the number of active nodes is computed at the same time in the *next* round. The analysis message increases its computation number (`N + 1`) which denotes which round the computation was done, since the number of active nodes are computed once every round.

For example, to count the number of active nodes in a network with 100 nodes, at the very end of each round for 10 rounds, we issue the command:

```
(tfrew {genInitConf(1, 100)
        dly(activeNodes(0,0),roundTime - 1)}
    in time < roundTime * 10 .)
```

Computing the percentage of coverage

The same approach is used to compute the percentage of coverage:

```
eq {coveragePercentage(N, 0) C} =
   {dly(coveragePercentage(N, coveragePercentage(C)), INF)
     dly(coveragePercentage(N + 1, 0), roundTime) C} .
```

The percentage of coverage in the sensing area is defined in [25] as the share of covered bits in the bitmap compared to the total number of bits. It is computed by the `coveragePercentage` function which traverses the sensing area bitmap, and divides the number of covered bits by the total number of bits.

Performing the analysis

We generate different initial states with varying number of deployed nodes with the `genInitConf` function and include the appropriate analysis messages in the initial state:

```
{genInitConf(seed , numNodes)
 dly(activeNodes(0,0),roundTime - 1)
 dly(coveragePercentage(0,0),roundTime - 1)}
```

This measures both the number of active nodes and the coverage percentage at the very end of each round.

We are interested in how the number of active nodes and coverage varies with different numbers of deployed nodes. We, therefore, only need to execute the simulation for one round:

```
(tfrew {genInitConf(seed, numNodes)
         dly(activeNodes(0,0),roundTime - 1)
         dly(coveragePercentage(0,0),roundTime - 1)}
     in time < roundTime .)
```

The simulations in [25] varies the number of deployed nodes from 100 to 1000 in a $50m \times 50m$ sensing area. As an example, the following `tfrew` command runs a simulation with 400 nodes. The size of the sensing area (`sensorAreaSize`) is $50m \times 50m$:

```
Maude> (tfrew {genInitConf(1, 400)
               dly(activeNodes(0,0),roundTime - 1)
               dly(coveragePercentage(0,0),roundTime - 1)}
           in time < roundTime .)


Result ClockedSystem :
{dly(coveragePercentage(0,100),INF)
dly(coveragePercentage(1,0),1000000)
dly(workingNodes(0,60),INF)
dly(workingNodes(1,0),1000000)
< Random : RandomNGen | seed : 5943 >

< 38 . -852 : WSNode |
backoffTimer : INF,
bitmap :(
        | ' ' ' ' ' ' ' ' ' ' t ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' t t t t t t t t t ' ' ' ' ' ' |
        | ' ' ' ' t t t t t t t t t t t t t ' ' ' ' |
        | ' ' ' t t t t t t t t t t t t t t t ' ' ' |
        | ' ' t t t t t t t t t t t t t t t t t ' ' |
        | ' ' t t t t t t t t t t t t t t t t t ' ' |
        | ' t t t t t t t t t t t t t t t t t t t ' |
        | ' t t t t t t t t t t t t t t t t t t t ' |
        | ' t t t t t t t t t t t t t t t t t t t ' |
        | ' t t t t t t t t t t t t t t t t t t t ' |
        | t t t t t t t t t t t t t t t t t t t t t |
```

```
| ’ t t t t t t t t t t t t t t t t t t t ’ |
| ’ t t t t t t t t t t t t t t t t t t t ’ |
| ’ t t t t t t t t t t t t t t t t t t t ’ |
| ’ t t t t t t t t t t t t t t t t t t t ’ |
| ’ ’ t t t t t t t t t t t t t t t t t ’ ’ |
| ’ ’ t t t t t t t t t t t t t t t t t ’ ’ |
| ’ ’ ’ t t t t t t t t t t t t t t t ’ ’ ’ |
| ’ ’ ’ ’ t t t t t t t t t t t t t ’ ’ ’ ’ |
| ’ ’ ’ ’ ’ ’ t t t t t t t t t ’ ’ ’ ’ ’ ’ |
| ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ t ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ |),
neighbours :((-181 . -2251 starting true)
             (-1279 . 97 starting false)
             (248 . -230 starting false)
             (148 . -21 starting false)
             (-833 . -1405 starting false)
             (-990 . -1515 starting false)
             (-1092 . -1514 starting false)
             (-526 . -1463 starting false)
             (686 . -1955 starting false)
             (-1034 . -1250 starting false)
             (1149 . -1848 starting false)
             (1151 . -1640 starting false)
             (-1191 . -1201 starting false)
             (-1342 . -1296 starting false)
             (1150 . -1903 starting false)
             (947 . -1797 starting false)
              1720 . -1277 starting false),
remainingPower : 1999000001,
roundTimer : 5/2,
status : off,
volunteerProb : 1,
hasVolunteered : false >

[...] } in time 999999
```

The result can be found in the result messages:

```
dly(coveragePercentage(0,100),INF)
dly(workingNodes(0,60),INF)
```

Of the 400 nodes that were deployed, 60 became active nodes and provide 100% coverage. More `tfrews` was executed with the same amount of nodes but with different seeds in the initial state, which resulted in an average of about 50 active nodes and all provided 100% coverage.

We want to investigate how the number of active nodes and percentage of coverage varies with respect to the number of deployed nodes. Therefore, we change the number of nodes in the initial state (second argument of the `genInitConf` function), and execute the same `tfrew` command. Starting with an initial state with 200 nodes resulted in 45 active nodes and 300 in 35 active nodes. All simulations showed 100% coverage. The results from the simulations in [25] show that the number of active nodes in the network is between 15 and 20 nodes, and provides 100% coverage when 500 or more nodes are deployed. When less nodes was deployed, 98-99% coverage was provided.

The simulations in [25] show that the number of active nodes is independent of the number of nodes that is deployed. This seems also to be the case in my simulations, though the number of active nodes varied quite a bit. However, the number of active nodes is *higher* in my simulations.

I have only executed a few simulations, but there is a clear tendency that they result in a somewhat higher number of active nodes. A factor that affect the number of active nodes is the number of nodes that volunteers to be starting nodes. The number of volunteering nodes in my simulations are usually between 1 to 5. If this is a larger number of volunteers than what was generally the case in the simulations in [25], this could explain some of the deviation. The higher number of active nodes will have an impact on the rest of the performance metrics as well.

Only a few simulations were performed in this example, where the seed was changed manually. It is possible, using Real-Time Maude's meta-level capabilities, to define a function which executes more timed rewrites automatically with different seeds, as outlined in Section 5.2.4.

5.2.2 The percentage of coverage and total amount of remaining power with respect to time

In [25] they are interested in how coverage and the total remaining power changes over time. These are important measurements, as they will show how long the network can stay alive and operational. This is performed in Real-Time Maude using the analysis messages to compute the percentage of coverage and the total remaining power once every round of the OGDC algorithm.

Computing the total remaining power

The percentage of coverage is computed as explained in the previous section. The total amount of remaining power is measured using the same technique as before:

```
eq {totalRemainingPower(N, 0) C} =
   {dly(totalRemainingPower(N, calcTotalRemainingPower(C)), INF)
    dly(totalRemainingPower(N + 1, 0), roundTime - 1) C} .
```

The function `calcTotalRemainingPower` traverses the configuration and sums up the value of each node's `remainingPower` attribute. The amount of remaining power is, as the other metrics, computed at the end of each round. However, the total remaining power can be computed at any time, since it does not require the network to be in the steady state phase. The reason it is computed at the end of the round is so it can be done at same time as the percentage of coverage, to avoid "stopping" time twice to compute these metrics.

Performing the analysis

Since we are interested in how coverage and total remaining power changes over time, the timed rewrite is executed for several rounds. In the simulations in [25], the last node dies after about 90 rounds of the OGDC algorithm. To capture the whole lifetime of the network, the timed rewrites are executed for 100 rounds. The number of nodes in these simulations are reduced to 300 in [25], but still placed in $50m \times 50m$ sensing area, which corresponds to 0.12 nodes per m^2. Because of the long execution time of a large amount of nodes for several rounds, I use 75 nodes in a $25m \times 25m$ area, which is a scale of $1 : 4$. The reason why it is reduced to 300 nodes in [25] is not known to us, but it can potentially be for the same reason, namely, to reduce the execution time.

```
Maude> (tfrew {genInitConf(1, 75)
               dly(coveragePercentage(0, 0),roundTime - 1)
               dly(totalRemainingPower(0, 0),roundTime - 1)}
           in time <= roundTime * 100 .)
```

```
Result ClockedSystem :
{dly(coveragePercentage(0,100),INF)
dly(coveragePercentage(1,100),INF)

[...]

dly(coveragePercentage(15,96),INF)
dly(coveragePercentage(16,100),INF)
dly(coveragePercentage(17,100),INF)
dly(coveragePercentage(18,69),INF)
dly(coveragePercentage(19,100),INF)
dly(coveragePercentage(20,95),INF)
dly(coveragePercentage(21,87),INF)
dly(coveragePercentage(22,93),INF)

[...]

dly(coveragePercentage(34,34),INF)
dly(coveragePercentage(35,0),INF)
dly(coveragePercentage(36,0),INF)

[...]


dly(totalRemainingPower(0,146733063635),INF)
dly(totalRemainingPower(1,141471269862),INF)

[...]

dly(totalRemainingPower(32,1169933192),INF)
dly(totalRemainingPower(33,771915788),INF)
dly(totalRemainingPower(34,373499783),INF)
dly(totalRemainingPower(35,0),INF)

[...]

 } in time 99999999
```

We can see from the result messages that the nodes can provide 100% coverage for 20 rounds. The reason for the decrease of percentage of coverage in round 15 and 18 is that one or more nodes died in the middle of those rounds. This causes the percentage of coverage to be temporarily decreased until the start of the next round, because new active nodes are not chosen before this time. This means that the OGDC algorithm does not always provide 100% coverage even if there exists enough "alive" nodes to do so. The last node in the network dies in round 34. Simulations with different seeds showed 100% coverage for up to 26 rounds, and the last node dies after 36 rounds. In [25] the nodes provide 100% coverage for about 40 rounds, and the last node dies close to round 90.

The difference between my results and the results achieved in [25] is significant. However, the results of the number of active nodes in Section 5.2.1 explains some of the deviating results,

since the more nodes that are active each round, the more power is consumed in the network. The shorter lifetime is, therefore, a natural consequence of the higher number of active nodes in each round.

5.2.3 α-lifetime with respect to α and with respect to the number of deployed nodes

α-lifetime is defined in [25] to be the total time during which at least α percent of the total sensing area is covered by at least one node. I have some difficulties interpreting this definition as it is also stated that it is slightly different from the definition in [23], where the lifetime is defined as the time interval until which coverage falls below a pre-determined percentage and never comes back up again. I therefore, use the `coveragePercentage` analysis message to estimate an *interval* in which the percentage of coverage is below α.

Performing the analysis

The simulations in [25] first investigates how α lifetime changes with various values of α. In [25] they use 300 nodes in a $50m \times 50m$ sensing area for α values between 50% and 100%. I use the same scale as in the previous section with 75 nodes in a $25m \times 25m$ area, and the `coveragePercentage` analysis message. The different α lifetimes can be found by reviewing the result messages:

```
Maude> (tfrew {genInitConf(325, 75)
               dly(coveragePercentage(0, 0),roundTime - 1)}
           in time <= roundTime * 100 .)

Result ClockedSystem :
{dly(coveragePercentage(0,100),INF)
dly(coveragePercentage(1,100),INF)
dly(coveragePercentage(2,100),INF)

[...]

dly(coveragePercentage(21,91),INF)
dly(coveragePercentage(22,90),INF)
dly(coveragePercentage(23,83),INF)
dly(coveragePercentage(24,71),INF)
dly(coveragePercentage(25,80),INF)
dly(coveragePercentage(26,54),INF)
dly(coveragePercentage(27,71),INF)
dly(coveragePercentage(28,0),INF)
dly(coveragePercentage(29,0),INF)

[...]
} in time 100000000
```

By reviewing the result messages we can estimate, for instance, the 75% lifetime to about 24 to 25 rounds. More simulations were run with different seeds which resulted in a 90% lifetime between 16 and 22 rounds, 75% lifetime between 21 and 29, and 50% lifetime between 26 and 30 rounds. The results show, as expected, that the lifetime of the network increases when decreasing the amount of percentage of coverage. The simulations in [25] showed an average of 90% lifetime

of about 65 rounds, 75% lifetime of 80, and 50% lifetime of 90 rounds. The results differ quite a bit, but is also possibly a result of the higher number of active nodes in each round.

The simulations in [25] secondly investigates how α lifetime changes, for a fixed α, when increasing the number of deployed nodes. More deployed nodes should result in a longer α lifetime, since the density of nodes increases and consequently more nodes can conserve power. The simulations in [25] investigates 90% and 98% lifetime with the number of deployed nodes varying from 100 to 800 in a $50m \times 50m$ sensing area. I scale down to about $1 : 6$ and operate on a $20m \times 20m$ area. The `tfrew` command is the same as before, and we vary the number of deployed nodes by changing the second argument of the `genInitConf` function. First we simulate 20 nodes in the $20m \times 20m$ area, which corresponds to 125 nodes in the $50m \times 50m$ area:

```
Maude> (tfrew {genInitConf(35, 20)
               dly(coveragePercentage(0, 0),roundTime - 1)}
           in time <= roundTime * 100 .)

Result ClockedSystem :
{dly(coveragePercentage(0,100),INF)
dly(coveragePercentage(1,100),INF)
dly(coveragePercentage(2,100),INF)
dly(coveragePercentage(3,100),INF)
dly(coveragePercentage(4,100),INF)
dly(coveragePercentage(5,87),INF)
dly(coveragePercentage(6,97),INF)
dly(coveragePercentage(7,99),INF)
dly(coveragePercentage(8,91),INF)
dly(coveragePercentage(9,50),INF)
dly(coveragePercentage(10,93),INF)
dly(coveragePercentage(11,86),INF)
dly(coveragePercentage(12,88),INF)
dly(coveragePercentage(13,63),INF)
dly(coveragePercentage(14,0),INF)
dly(coveragePercentage(15,0),INF)

[...]
} in time 100000000
```

I only consider 90% lifetime as 98% lifetime can be found analogously. 90% lifetime is achieved for about 9 rounds which is very close to the result they found in the simulations in [25]. I also did simulations with 64 (corresponds to 400 nodes in a $50m \times 50m$ area), 80 nodes (500), and 125 nodes (800), which resulted in a 90% lifetime of between 24 and 28 rounds, 32 to 36, and 45 to 50 rounds, respectively. In [25] they achieved about 65, 85, and 140 rounds, respectively, for these number of nodes. The deviations in results are quite large. However, my simulations, as well as the simulations in [25], show that the lifetime is prolonged when more nodes are deployed. The deviation might still be a result of the higher number of active nodes in each round. The results deviate more when increasing the number of deployed nodes. This is a natural consequence of the higher number of active nodes in each round, because when more nodes are deployed, the longer the network stays "alive", and the more rounds are affected by the higher number of active nodes. This, therefore, explains the increasing deviation. However, whether the higher number of active nodes is the only reason for the shorter lifetimes is uncertain.

5.2.4 Iterative simulations using Real-Time Maude's meta-level

Real-Time Maude incorporates reflective and meta-programming features, which allows for user defined search and simulation strategies on the meta-level. Iterative simulations can be achieved by specifying a new simulation strategy on the meta-level, which loops and runs several `tfrew` commands starting with different initial states.

5.3 Further analysis of the OGDC algorithm in Real-Time Maude

This section explains the further analysis of the OGDC algorithm using Real-Time Maude's *timed search*, *temporal logic model checking*, and the time-specific searches. As mentioned in the beginning of this chapter, this form of analysis investigates *all* possible behaviors in a system (with respect to the selected time sampling strategy). The number of states grows exponentially in the OGDC model when we add more nodes to the initial states, but the state space is *finite* when we analyze with a certain time bound. Provided with enough computer memory (and human patience), the whole state space can be explored in finite time.

Since this type of analysis performs the time consuming task of investigating every possible behavior starting from the initial state, the analysis is done with a small number of nodes to reduce the state space and thereby the execution time. However, analysis of other algorithms and protocols in Maude and Real-Time Maude has previously shown that it is often not required to do analysis on large initial states to find significant errors [18, 9].

The analysis performed in this section investigates some of the essential parts of the OGDC algorithm. All the nodes in the initial states are placed within a $15m \times 15m$ sensing area, unless stated otherwise, to provide a fair chance of covering the sensing area and of getting sufficient overlap of the nodes 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. The sensing task in the OGDC algorithm 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 networks lifetime is used for sensing.

We can use Real-Time Maude's `find latest` command to find the latest possible time the network enters the steady state phase, and thereby also find out whether the phase is always reached within the end of the round.

If a node does not volunteer as a starting node, its backoff timer is set to the constant `nonVolunteerTimer` (1 second). Therefore, in the case where no nodes volunteers at the start of the round, it takes 1 second until the nodes try again. This adds one second to the total time it takes to reach the steady state phase. Another second is added each time none of the nodes volunteer after that. At least one node will eventually volunteer, because the volunteer probability is doubled each time, which kicks off the selection of active nodes. We wish to investigate how long time the actual selection of active nodes takes. We, therefore, investigate the time it takes from at least *one* node volunteers until the network is in the steady state phase.

The initial state below contains 5 nodes where the random number generator (that is, the `RandomNGen` object created with the `genInitConf` function) causes one node to volunteer to be a starting node (the other four nodes do not volunteer), at the beginning of the round. The `find latest` command checks all possible states which has reached the steady state phase (`such that steadyStatePhase(...)`) and returns the state which is reached after the longest period of time.

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

Result: { [...] } in time 1014
```

The latest possible time the network enters the steady state phase from this initial state is after 1014 milliseconds. One round is set to 1000 *seconds* which means that the network spends at least about 999 of those seconds in the steady state phase performing its sensing task. If none of the nodes volunteer at the beginning of the round, a second is added the total time, but the greater part of the networks lifetime is still spent performing its sensing task.

Since the result is the *latest* time the steady state is reached, this also implies that the steady state phase is *always* reached within the end of the round from this initial state.

The previous search found the "worst" possible scenario. We can also find the earliest time the network enters the steady state phase with the `find earliest` command:

```
Maude> (find earliest {genInitConf(1, 5)} =>* {C:Configuration}
                   such that steadyStatePhase(C:Configuration) .)

Result: { [...] } in time 157
```

The earliest time the network enters the steady state phase is after 157 milliseconds. This means that all behaviors, starting from this initial state, has reached the steady state phase at some time within the time interval from 157 to 1014 milliseconds. These two searches were performed on more initial states with different seeds with similar results. The time it takes from at least one node volunteers until the steady state phase is reached, seems to be at least less than about one second.

It is stated in [25] that the steady state phase can be reached in well below 1 second and often in less than 200 milliseconds. We investigate this claim by finding how many states are in the steady state phase before 200 milliseconds. This is found using a timed search within a time interval, by searching for all states (any state matching `{ C:Configuration }`) that has reached the steady state phase (`such that steadyStatePhase(...)`) at time 200 (`time-interval between >= 200 and <= 200`). With the current time sampling strategy the time 200 milliseconds might not be visited. To be able to investigate this exact time we must change the time sampling strategy:

```
(set tick def 1 .)
```

We search for all states at time 200 that have reached the steady state phase:

```
Maude> (tsearch {genInitConf(13, 5)} =>* {C:Configuration}
             such that steadyStatePhase(C:Configuration)
             in time-interval between >= 200 and <= 200 .)

[...]

Solution 68
C:Configuration <- < Random : RandomNGen ...

[...]
```

To find the share of states that has reached the steady state phase we must find all states that are reachable at time 200:

```
Maude> (tsearch {genInitConf(13, 5)} =>* {C:Configuration}
             in time-interval between >= 200 and <= 200 .)

[...]

Solution 163
C:Configuration <- < Random : RandomNGen ...

[...]
```

Out of the 163 states that are reachable at 200 milliseconds, 68 of them are states which are in the steady state phase. However, this result does not say how many *behaviors* that have reached the steady state phase, since several behaviors can end up in the same state. The states that has reached the steady state phase probably represent more behaviors than the states that have not reached this phase, because a node's attributes are set to specific values when switches on or off and, therefore, there is a bigger chance that more states coincide when every node has been switched on or off.

Another interesting property to investigate is if the network stays in the steady state phase for the whole round, once this phase has been reached. Timed search can *not* be used to investigate this property. However, we can use model checking by defining a temporal logic proposition:

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

which is defined to `true`, when the network is in the steady state phase. The proposition is checked in every state, in every possible behavior, starting from the initial state. The temporal logical formula checks whether all states following a state which is in the steady state phase is also in this phase[1].

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

Result Bool :
  true
```

The temporal logical formula is true, that is, the network stays in the steady state phase once this phase has been reached. The same result was found for other initial states.

5.3.2 Coverage

We found from the simulations in Section 5.2.2, that the sensing area is not necessarily covered even though there are enough "alive" nodes to cover it. However, the sensing area should be covered in the steady state phase when there are enough nodes in the network to cover it and all nodes have enough power to last the round.

We define a proposition `sensing-area-covered`:

```
eq {C} |= sensing-area-covered = sensingAreaCovered(updateArea(sensingArea, C)) .
```

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

The constant `sensingArea` is defined in the `OGDC-ANALYSIS` module to be the sensing area bitmap of size `sensingAreaSize`. The function `updateArea` updates the bitmap by changing bits that are covered by the *active* nodes in the configuration `C` to `t`. The function `sensingAreaCovered` traverses the bitmap and returns `true` if, and only if, all the bits in the bitmap is set to `t`.

The nodes in the initial state can, potentially cover the sensing area, but not all are required to be active, and all the nodes have enough power to last the whole round.

```
Maude> (mc {genInitConf(5,5)} |=t (steady-state-phase => sensing-area-covered)
             in time <= roundTime .)


Result Bool :
  true
```

The sensing area is covered in the steady state phase in all behaviors starting from this initial state. The result was the same for other similar initial states.

5.3.3 A node's status and coverage area

A key aspect of the OGDC algorithm is that a node is switched off when its coverage area is covered. An expected unwanted state is, therefore, a state where a node is switched *on* even though its coverage area *is* covered. We define a function `isCovered?` that 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.

We search for a state which contains a node (`< O:Oid : WSNode ...>`) with its status attribute set to `on`, and a configuration (`C:Configuration`) that causes the coverage area of the node to be covered (`such that isCovered(...)`). We search only within the first round of the protocol:

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


Solution 1
O:Oid <- 505 . -252 ;
ATTS:AttributeSet <-
        backoffTimer : INF,
        bitmap :(
        | ' ' ' ' ' ' ' ' ' ' f ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' f f f f f f f ' ' ' ' ' ' ' ' |
        | ' ' ' ' f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' ' ' f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' ' f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' ' f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | f f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' f f f f f f f f f f f f ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
```

```
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |),
        neighbours : nil,
        remainingPower : 1999903197,
        roundTimer : 999790,
        volunteerProb : 20, hasVolunteered : true ;

C:Configuration <-
< 462 . -224 : WSNode |
        backoffTimer : INF,
        bitmap :(
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' f t t t t t t ' ' ' ' ' ' ' ' |
        | ' ' ' ' f t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' ' ' f t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' ' f t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' ' t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' f t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' t t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' t t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' t t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | f t t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' t t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' t t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' t t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' f t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' ' t t t t t t t t t t t ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |
        | ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' |),
        neighbours : 505 . -252 starting true,
        remainingPower : 1999985790,
        roundTimer : 999790,
        status : on,
        volunteerProb : 20,
        hasVolunteered : false >

[...]
```

The result of the search is a state where node 505 . -252 is switched on, even though the rest of the active nodes cover its coverage area. This means that this node can be switched off without affecting the percentage of coverage the network provides. The number of active nodes can, therefore, be reduced by switching this node off. The reason that this node has not been switched off is because its coverage area was covered *after* it had been switched on. When a node is switched on it ignores all power-on messages, and consequently does not update its coverage area when surrounding neighbors switch on.

My first intuition was that switching off these “redundant” active nodes could easily be achieved, by having each active node *not* ignoring power-on messages, but update its coverage area and switch off if its coverage area becomes completely covered. A parallel model with this modification was created, to investigate further. After a few prototyping simulations the

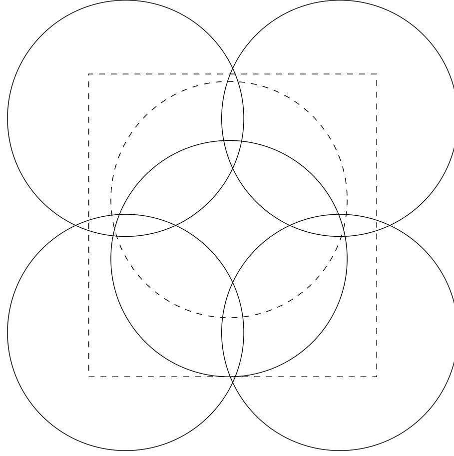


Figure 5.1: A tailored initial state for checking coverage when a node dies.

alternate model produced an unwanted result. A state that did not provide full coverage was found in the first round, even though the nodes in the initial state could potentially cover the sensing area, and all the nodes had enough power for the whole round. The reason for this was found to be the following. Consider the case where two neighboring nodes are active, and one of the nodes switches off because its coverage area becomes covered. This node does not notify its neighbor that it is switched off. The neighbor, therefore, still thinks that the part of the coverage area that *was* covered, by the nodes that now is switched off, is *still* covered. If the neighbor gets the rest of its coverage area covered, it is switched off even though the part of its coverage area that was covered by the node that is now switched off is no longer covered.

This search shows that there exists behaviors in the system where more nodes than necessary are active. However, to decide whether an already *active* node can be switched off is not a trivial task. This is something the developers of the OGDC algorithm obviously already have considered, and decided that this feature requires too much extra administration to be worthwhile. Additionally, this will require extra messages to be broadcasting, which consumes more energy, and it causes an unnecessary “unstable” network topology in the steady state phase, which is undesirable.

The extra complications connected with nodes being switched off in the middle of a round is also a reason for the OGDC algorithm to have a power threshold for volunteering to be a starting node. When a node dies in the middle of a round, the situation described above may occur. Therefore, preventing the nodes that have only a small chance of making it through the whole round alive to volunteer, postpones this situation until it is inevitable.

5.3.4 When nodes die

Since state space exploration over several rounds is an enormous task because of combinatorial explosion, we tailor an initial state to investigate the behaviors of the system when nodes start dying. The following initial state `init` contains six nodes placed as showed in Figure 5.1. The dotted square is the sensing area of size $25m \times 25m$, the solid circles are nodes that are active, and the dotted circle is a node that is switched off, because its coverage area is covered. The *active* node in the middle has only enough power for to stay alive for half a round, that is, this node will die by the end of the round. Since there is still enough nodes to cover the sensing area after this node is dead, we would like to investigate whether the sensing area is in fact *always* covered in all possible behaviors in the next round.

We search for a state starting from `init` which has reached the steady state phase, but where the sensing area is not covered. Since the steady state phase is usually reached within a few seconds, we search in the interval from the start of the next round and until 10 000 milliseconds:

```
Maude> (tsearch [1] init =>* {C:Configuration}
             such that steadyStatePhase(C:Configuration) and
                       not checkCoverage(checkArea(sensorArea, C:Configuration))
             in time-interval between > roundTime and <= roundTime + 10000 .)

No solution
```

No solution was found, which means the sensing area is always covered when the network is in the steady state phase, in all behaviors starting from the tailored initial state `init`. The same search from other similar initial states gave the same result.

5.4 Concluding remarks of the analysis

We now sum up the results of the simulations and state space exploration analysis performed in Real-Time Maude.

All the simulations done using ns-2 in [25] could be performed using Real-Time Maude without any problems. The results of the simulations showed a higher number of active nodes compared to the simulations in [25]. The reason for this is not evident. A factor that affects the number of active nodes is the number of volunteering nodes. Although we did not get a particularly high number of volunteering nodes (usually under 5), if the simulations in [25] resulted in usually just 1 or 2 volunteers, this can explain some of the difference in the results. The network also had a constantly shorter lifetime in our simulations than in [25]. The higher number of active nodes, naturally, affects the lifetime of the network, and the increasing deviation in lifetimes between our simulations and those in [25] when running more rounds suggests that this is a factor. However, there might be other additional reasons for the constantly shorter lifetime.

In the further formal analysis, we checked all possible behaviors in one round starting from some chosen initial states. We found that the steady state phase is reached in an early stage of each round, as claimed in [25]. We discovered that there exists behaviors where more nodes than necessary become active, but that this most likely requires too much administration to be worthwhile. Other than this, all other state space explorations gave expected results, and no errors were found.

Chapter 6

Conclusion

In this thesis, I have looked at how the OGDC algorithm for wireless sensor networks can be formally modeled, simulated, and formally analyzed in Real-Time Maude. Wireless sensor networks and the OGDC algorithm poses a diverse list of challenges that has been met by Real-Time Maude as explained. We managed to perform all the simulations that Zhang and Hou did using ns-2 in [25][1]. Additionally, we could perform further formal analysis by investigating all possible behaviors with time bounded search and model checking, including properties like whether coverage is *always* achieved in the steady state phase, and whether alive nodes *always* cover up for nodes that die.

The expressive and general formalism of Real-Time Maude seemed well suited to formally model the new aspects introduced by the wireless sensor network domain and the OGDC algorithm. Broadcast communication with limited range is naturally modeled, and the other aspects of the OGDC algorithm, like power consumption, are integrated without problems. The executable specification was relatively quickly modeled because of Real-Time Maude's flexible and intuitive formalism. The formalization process helped discover ambiguities in the informal description, which were made explicit in the model, resulting in a precise specification of the OGDC algorithm at a fairly high level of abstraction.

In the analysis section, we could perform simulations analogous to those done using ns-2 in [25] without any problems. However, the execution of the simulations went slower than expected[2], so the simulations were performed with fewer nodes than in [25]. The area was also reduced to keep the same density of nodes as in [25]. The focus of the specification is clarity rather than optimization, but the reason for these execution times should be investigated further, as time did not allow for it by the completion of this thesis. The simulations using ns-2 in [25] use 300 nodes when simulating over more than one round. The reason for reducing the number of nodes is unknown to us, but it might be due to long execution times, though this is pure speculation.

The simulations of the Real-Time Maude specification resulted in about the same number of active nodes regardless of the number of nodes that were deployed, as in [25], but the number of active nodes were generally *higher* than showed in the ns-2 simulations in [25]. This affected the other simulation results, which, naturally, diverted more from the results in [25] as the simulations went over more rounds. The exact reason for this higher number of active nodes is not apparent. The number of nodes that volunteer to be starting nodes affects the number of active nodes. The number of volunteers in my simulations was generally between 1 to 5. If the simulations in [25] generally had 1 to 2 volunteers, this explains some of the deviation. The information about the number of volunteers in [25] is not known to us, so this is also pure

[1] This only includes the simulations done under the listed assumptions in Section 4.2 and with 1-coverage.

[2] A simulation of 1000 nodes in a $50m \times 50m$ area for one round took almost 20 hours.

speculation.

In addition to the simulations, we could perform further formal analysis by investigating *all* behaviors from an initial state. We could investigate properties that can not be checked using only simulations. Because of the combinatorial explosion when adding more nodes, we used 5 or 6 nodes in the state exploring analysis, and used both random generated and specifically tailored initial states to investigate some performance metrics, and search for design errors in the OGDC algorithm. Search and model checking with few number of nodes have, as mentioned, previously found subtle design errors in other communication protocols, which shows that this kind of analysis is highly useful even with few nodes. We had good use of Real-Time Maude's `find earliest` and `find latest` commands to find the extremities of the time it takes to reach the steady state phase, and used search and model checking to investigate other properties. No significant errors were found in this state exploring analysis, which is somewhat surprising, considering other case studies done in Real-Time Maude [9, 18]. This, however, shows that the developers, Zhang and Hou, have designed a solid algorithm, and that the OGDC algorithm has been well tested.

To conclude, it seems that Real-Time Maude's flexible formalism is well suited to formally model the combination of aspects introduced by wireless sensor networks, and complements existing network targeted simulation tools by offering an high-level intuitive formal specification and additional analysis features.

Future Work

It is desirable to find the reason for the long execution time of the simulations. That way we can run simulations with more nodes, so we don't have to scale down when running simulations over several rounds. The reason for the higher number of active nodes should also be investigated further, as there might be other, or additional, explanations than a possible higher number of volunteers in my simulations than in the simulations in [25].

Bibliography

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

[2] Rimon Barr, Zygmunt Haas, and Robert van Renesse. Jist: An efficient approach to simulation using virtual machines. *Software Practice and Experience*, 2004.

[3] 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.

[4] Nirupama Bulusu, John Heidemann, and Deborah Estrin. Gps-less low cost outdoor localization for very small devices. *IEEE Personal Communications Magazine*, 7(5):28–34, October 2000.

[5] Nirupama Bulusu, John Heidemann, Deborah Estrin, and Tommy Tran. Self-configuring localization systems: Design and experimental evaluation. Technical Report 8, University of California, Los Angeles, Center for Embedded Networked Computing, September 2002. Accepted to appear, ACM TOCS.

[6] 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.

[7] M. Clavel, F. Dúran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1)*, April 2004. `http://maude.cs.uiuc.edu`.

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

[9] 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.

[10] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, 2002.

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

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

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

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

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

[16] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of real-time maude. In *Workshop on Rewriting Logic and its Applications, Preliminary Version.*

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

[18] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Technical Report UIUCDCS-R-2004-2467, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004. Available at `http://www.ifi.uio.no/RealTimeMaude`.

[19] Real-Time Maude home-page. `http://heim.ifi.uio.no/~peterol/RealTimeMaude/`.

[20] Andreas Savvides, Chih-Chieh Han, and Mani B. Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Mobile Computing and Networking*, pages 166–179, 2001.

[21] An swol Hu and Sergio 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.

[22] Brett A. Warneke and Kritofer S.J. Pister. Mems for distributed wireless sensor networks. In *9th International Conference on Electronics, Circuits and Systems*. Berkley Sensor and Actuator Center, University of Californie at Berkley, 2002.

[23] F. Ye, G. Zhong, S. Lu, and L. Zhang. Energy efficient robust sensing coverage in large sensor networks. Technical report, UCLA, 2002.

[24] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. Glomosim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

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

Appendix A

The 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's no starting neighbour and
    *** O' is the closest neighbour .
    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***
    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 ****************
******************************************

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

```
sorts Location Status VolunteeredStatus NeighbourList Neighbour .

subsort Location < Oid .
subsort Neighbour < NeighbourList .
subsort Bool < VolunteeredStatus .

***Represent a node by it's coordinate
op _._ : Rat Rat -> Location [ctor] .


***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
op undecided : -> VolunteeredStatus [ctor] .


***A neighbour is represented by it's coordinate and if it's a starting node
op _starting_ : Location Bool -> Neighbour .
op nil        : -> NeighbourList [ctor] .
op __         : NeighbourList NeighbourList -> NeighbourList [ctor assoc id: nil] .

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


***( Auxiliary functions )***
op isInSensingArea : Location -> Bool .

op vectorLengthSq : Location Location -> Rat .

op _withinSensingRangeOf_ : Oid Oid -> Bool .
op _withinTwiceOfSensingRangeOf_ : Oid Oid -> Bool .
op _withinTransmissionRangeOf_ : Oid Oid -> Bool .


vars O O' : Oid .
vars X X' Y Y' : Rat .
var N : Nat .

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


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

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

    eq O withinTwiceOfSensingRangeOf O' = vectorLengthSq(O, O')
                                           <= (4 * sensingRange * sensingRange) .

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


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 grid increment
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 : Oid Location Nat Nat -> Bitmap .
op makeRowBitmap : Oid Location Nat Nat -> BitList .


***( Update bitmap )***
op updateBitmap : Oid Bitmap Oid -> Bitmap . ***BMOwner BM NewNeighbour
op updateBitmap : Bitmap Oid Location Nat -> Bitmap .
op updateBitList : BitList Oid Location Nat -> BitList .


***( Check if sensor area is completely covered )***
op sensingAreaCovered : Oid Bitmap Oid -> Bool . ***BMOwner BM NewNeighbour
op checkCoverage : Bitmap -> Bool .



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


***************
***Equations***
***************


*** 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 O, with N rows and columns and increment I
    eq makeColumnBitmap(O, X . Y, I, N) =
        if N > 0
        then | makeRowBitmap(O, X . Y, I, gridCounter) |
       makeColumnBitmap(O, 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, O') = updateBitmap(BM, O', X - sensingRange .
                                                  Y + sensingRange, gridInc) .

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

    *** Update one row of the bitmap
    eq updateBitList(nil, O, L, I) = nil .
    eq updateBitList(BIT BITL, O, X . Y, I) =
        if (BIT =/= f)
then BIT
        else if O withinSensingRangeOf (X . Y)
     then t
     else f
     fi
        fi
updateBitList(BITL, O, X + I . Y, I) .


    *** Check if sensing area is completely covered after updated
    *** with the latest neighbour's position
    eq sensingAreaCovered(O, BM, O') = checkCoverage(updateBitmap(O, BM, O')) .

    eq checkCoverage(nil) = true .
    eq checkCoverage(| nil | BM) = checkCoverage(BM) .
```

```
    eq checkCoverage(| BIT BITL | BM) = (BIT =/= f) and checkCoverage(| BITL | BM) .

endom)




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


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

    sorts Crossing CrossingList .

    subsort Crossing < CrossingList .

    ***( A crossing between two sensing areas is represented by
         the two nodes that creates the crossing and its coordinate )***
    op _x_in_ : Oid Oid Location -> Crossing [ctor] .
    op nil : -> CrossingList [ctor] .
    op __ : CrossingList CrossingList -> CrossingList [ctor assoc id: nil] .



    ***( Determine crossings (covered and uncovered) within range of a node )***
    op existsUncoveredCrossings : Oid NeighbourList -> Bool .
    op listUncoveredCrossings   : OidSet CrossingList -> CrossingList .
    op checkSingleCrossing      : OidSet Crossing -> Bool .
    op extractValidCrossings    : Oid CrossingList -> CrossingList .
    op findAllCrossings         : OidSet -> CrossingList .
    op findAllPairCrossings     : OidSet -> CrossingList .


    ***( Determine if a node creates the closest crossing point
    ***  to another node )***
    op _createsClosestCrossing__ : Oid Oid NeighbourList -> Bool .
    op closestCrossing           : Oid CrossingList -> Crossing .


    ***( Determine closest (starting or non-starting) neighbour  )***
    op existsStartingNeighbour      : NeighbourList -> Bool .
    op findClosestStartingNeighbour : Oid NeighbourList -> Oid .
```

```
op listStartingNeighbours       : NeighbourList -> OidSet .
op compareNeighbours            : Oid OidSet -> Oid .
op findClosestNeighbour         : Oid NeighbourList -> Oid .


***( Calculate a single crossing )***
op findPairCrossings  : Oid Oid -> CrossingList .
op calcLinearCrossing : Oid Oid Bool -> Location .

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


***( Other auxiliary functions )***
op extractOid                : NeighbourList -> OidSet .
op NodeCreatesCrossing       : Oid Crossing -> Bool .
op extractFirstCrossingNode  : Crossing -> Oid .
op extractSecondCrossingNode : Crossing -> Oid .
op extractCrossing           : Crossing -> Location .



var B                 : Bool .
vars O O' O'' O1 O1' : Oid .
vars OS               : OidSet .
vars L L'             : Location .
vars X X' Y Y' R      : Rat .
vars C                : Crossing .
vars CL               : CrossingList .
var NBL               : NeighbourList .


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


eq existsUncoveredCrossings(O, NBL) = listUncoveredCrossings(extractOid(NBL),
         extractValidCrossings(O, findAllCrossings(extractOid(NBL)))) =/= nil .


eq O' createsClosestCrossing O NBL = NodeCreatesCrossing(O',
closestCrossing(O,
  listUncoveredCrossings(extractOid(NBL),
    extractValidCrossings(O,
      findAllCrossings(extractOid(NBL)))))) .
```

```
    ***Find closest crossing
    ***NB! Caller has to ensure that CrossingList =/= nil
    eq closestCrossing(O, C) = C .
    eq closestCrossing(O, (O' x O'' in L) (O1 x O1' in L') CL) =
        if vectorLengthSq(O, L) < vectorLengthSq(O, L')
        then closestCrossing(O, (O' x O'' in L) CL)
        else closestCrossing(O, (O1 x O1' in L') CL)
        fi .


    *** Only keep crossings that are uncovered by a third node
    eq listUncoveredCrossings(OS, nil) = nil .
    eq listUncoveredCrossings(OS, (O x O' in L) CL) =
        if checkSingleCrossing(OS minus O ; O', (O x O' in L)) ***
        then (O x O' in L) ***
        else nil
        fi
        listUncoveredCrossings(OS, CL) .


    *** Determine if a crossing is within range of the nodes in OS
    eq checkSingleCrossing(none, C) = true .
    eq checkSingleCrossing(O ; OS, (O' x O'' in L)) =
        (not O withinSensingRangeOf L) and checkSingleCrossing(OS, (O' x O'' in L)) .



    ***Only keep the crossings within sensor range of O
    eq extractValidCrossings(O, nil) = nil .
    eq extractValidCrossings(O, (O' x O'' in L) CL) =
        if O withinSensingRangeOf L
then (O' x O'' in L) ***
else nil
fi
extractValidCrossings(O, CL) .

    *** Find all crossings made by the the nodes in OS
    eq findAllCrossings(none) = nil .
    eq findAllCrossings(O) = nil .
    eq findAllCrossings(O ; O' ; OS) =
        findAllPairCrossings(O ; OS)
        findAllCrossings(O' ; OS) .


    eq findAllPairCrossings(none) = nil .
    eq findAllPairCrossings(O) = nil .
    eq findAllPairCrossings(O ; O' ; OS) =
        (if O withinTwiceOfSensingRangeOf O'
         then findPairCrossings(O, O')
```

```
else nil
      fi)
      findAllPairCrossings(O ; OS) .



   *** Find the crossing(s) 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 two nodes is directly above and below eachother
   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) .


   *** Round off to nearst cm with use of ceiling
   eq y(O, O', true) = ceiling((- b(O, O') + delta(O, O')) / (2 * a(O, O'))) .
   eq y(O, O', false) = ceiling((- b(O, O') - delta(O, O')) / (2 * a(O, O'))) .

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


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




   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(O, O') = 1 + (u(O, O') * u(O, O')) .

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

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




*** Determine if a starting neighbout exists
eq existsStartingNeighbour(nil) = false .
eq existsStartingNeighbour((X . Y starting B) NBL) = B or
existsStartingNeighbour(NBL) .



*** Find the closest starting neighbour to O in NNL
*** NBL is never nil
eq findClosestStartingNeighbour(O, NBL) = compareNeighbours(O,
                                              listStartingNeighbours(NBL)) .


*** List all starting neighbours
eq listStartingNeighbours(nil) = none .
eq listStartingNeighbours((X . Y starting B) NBL) =
   if B
   then X . Y
   else none
   fi
   ; listStartingNeighbours(NBL) .


*** Find the closest neighbour to O in NBL
eq findClosestNeighbour(O, NBL) = compareNeighbours(O, extractOid(NBL)) .


*** Find the closest neighbour to O in OS
eq compareNeighbours(O, O') = O' .
eq compareNeighbours(O, O' ; O'' ; OS) =
    if vectorLengthSq(O, O') < vectorLengthSq(O, O'')
    then compareNeighbours(O, O' ; OS)
    else compareNeighbours(O, O'' ; OS)
    fi .


eq extractOid(nil) = none .
eq extractOid((O starting B) NBL) = O ; extractOid(NBL) .

eq NodeCreatesCrossing(O, O' x O'' in L) = O == O' or O == O'' .
```

```
    eq extractFirstCrossingNode(O x O' in O'') = O .
    eq extractSecondCrossingNode(O x O' in O'') = O' .
    eq extractCrossing(O x O' in O'') = O'' .



endom)





******************************************
***** BACKOFF TIMER COMPUTATIONS *********
******************************************


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


    ***( Timer calculations )*** These will always return positive values

    op setTC1 : Oid NeighbourList Nat -> Time .

    *** Angle between optimal location and location of receiver node
    op dAlphaTC1 : Location NeighbourList -> Rat . *** Sender Neighbours
    op dAlphaTC1 : Location Crossing -> Rat . *** Sender (Crossing created by sender)

    ***Distance from receiver to closest crossing point***
    op dTC1 : Oid NeighbourList -> Rat .



    op setTC2 : Oid Oid Nat NeighbourList Nat -> Time .

    *** Angle between desired node position and reveiver node
    op dAlphaTC2 : Location Location Nat -> Rat . *** Sender Receiver Direction

    ***Distance from receiver to sender***
    op dTC2 : Oid Oid -> 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 between two vectors in degrees
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 X X' Y Y' : Rat .
vars N D : Nat .
vars O O' : Oid .
var OS : OidSet .
var C : Crossing .
var CL : CrossingList .
var NBL : NeighbourList .
vars L : Location .



eq setTC1(O, NBL, N) = ceiling(transmissionDelay * (c *
    (((sensingRange - dTC1(O, NBL)) * (sensingRange - dTC1(O, NBL)))
    + (dTC1(O, NBL) * dTC1(O, NBL) * dAlphaTC1(O, NBL) * dAlphaTC1(O, NBL)))
   + randomU(N))) .


*** Distance between receiver node and the closest uncovered crossing point
eq dTC1(O, NBL) = sqrt(vectorLengthSq(O, extractCrossing(closestCrossing(O,
                                          listUncoveredCrossings(extractOid(NBL),
    extractValidCrossings(O,
                                      findAllCrossings(extractOid(NBL)))))))) .


***( Angle between the optimal node location and the location of
*** the receiving node )***
ceq dAlphaTC1(O, NBL) = dAlphaTC1(O, closestCrossing(O,
listUncoveredCrossings(OS,
  extractValidCrossings(O,
    findAllCrossings(OS)))))
   if OS := extractOid(NBL) .
```

```
ceq dAlphaTC1(O, X . Y x X' . Y' in L) =
(if R > pi
 then 2 * pi - R
 else R
 fi)
  if R := abs(angle(L, O) - angle(((X' + X) / 2 . (Y' + Y) / 2), L)) .
```

```
eq setTC2(O, O', D, NBL, N) = ceiling(transmissionDelay * (c *
    (((sqrt(3) * sensingRange - dTC2(O, O')) * (sqrt(3) * sensingRange - dTC2(O, O')))
    + (dTC2(O, O') * dTC2(O, O') * dAlphaTC2(O, O', D) * dAlphaTC2(O, O', D)))
   + randomU(N))) .
```

```
*** Distance from sender to receiver
eq dTC2(O, O') = sqrt(vectorLengthSq(O, O')) .
```

```
*** Angle between the desired direction for which the node should be
*** located and the direction from the sender to the receiver
ceq dAlphaTC2(O, O', D) = (if R > pi
 then 2 * pi - R
 else R
 fi)
  if R := abs(angle(O', O) - ((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(L)
  then 2 * pi - acos(dotProd(L, 1 . 0))
  else acos(dotProd(L, 1 . 0))
  fi)
if L := 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 .


    ***( Messages )***
    msg broadcastFrom_withDirection_ : Oid Int -> Msg .
    msg PowerOnMsgFrom_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,
                   neighbours : NeighbourList,
   remainingPower : Nat,
   roundTimer : TimeInf,
   status : Status,
                   volunteerProb : Rat .

endtom)



*****************************************
********* THE OGDC ALGORITHM *************
*****************************************


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


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

```
***( Auxiliary fuctions )***
*** Double the probability for volunteering
op doubleProb : Rat -> Rat .
*** 0, 100>
op randomProb : Nat -> Nat .
*** [0, td>
op randomTimer : Nat -> Time .
*** [0, 360>
op randomDirection : Nat -> Nat .




var B : Bool .
vars O O' : Oid .
var D : Int .
vars M N : Nat .
vars R P : Rat .
var L : Location .
var BM : Bitmap .
vars NB NEWNB : Neighbour .
var NBL : NeighbourList .
vars T : Time .
var S : Status .
var C : Configuration .
var MSG : Msg .


*******************
***( Equations )***
*******************


***( Message broadcasting )***
eq {(broadcastFrom O withDirection D) C} = {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' and O =/= O'
then dly((PowerOnMsgFrom O to O' withDirection D), transmissionDelay)
else none
fi .



    eq doubleProb(R) = if R >= 50
                       then 100
                       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 .




    ***************
    ***( Rules )***
    ***************

    ***( 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 == 1) *** with probability N
        then < O : WSNode | backoffTimer : randomTimer(random(M)), hasVolunteered : true >
        else < O : WSNode | backoffTimer : nonVolunteerTimer, hasVolunteered : false >
        fi)
        < Random : RandomNGen | seed : repeatRandom(M, 3) > .




    ***( If the ts timer expires, double the probability for volunteering,
    ***  and volunteer again )***
    rl [repeatVolunteering] :
       < O : WSNode | backoffTimer : 0, neighbours : nil, volunteerProb : R,
      hasVolunteered : false >
      =>
```

```
      < O : WSNode | backoffTimer : INF, volunteerProb : doubleProb(R),
                     hasVolunteered : undecided > .
```

```
    ***( 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) >
      broadcastFrom O withDirection randomDirection(M) . *** power on as starting node
```

```
    ***( Power on and send out a message when the timer expires )***
    rl [nonStartingNodePowerOn] :
      < O : WSNode | remainingPower : P, backoffTimer : 0, neighbours : NB NBL,
     hasVolunteered : false >
     =>
      < O : WSNode | remainingPower : P monus transPower, backoffTimer : INF, status : on >
      broadcastFrom O withDirection -1 . *** power on 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 neighbours completely cover its sensing area )***
    crl [recPowerOnMsgAndSwichOff] :
(PowerOnMsgFrom O' to O withDirection D)
< O : WSNode | status : S, neighbours : NBL, bitmap : BM, remainingPower : P >
     =>
      < O : WSNode | status : off, neighbours : NBL (O' starting (D >= 0)),
      bitmap : updateBitmap(O, BM, O'),
      backoffTimer : INF >
     if S =/= on /\ P > 0 /\
```

```
 sensingAreaCovered(O, updateBitmap(O, BM, O'), O') .




    ***( Update timer to TC1 if O' creates closest uncovered crossing )***
    crl [recPowerOnWithUncoveredCrossings] :
(PowerOnMsgFrom O' to O withDirection D)
< O : WSNode | remainingPower : P, status : S, backoffTimer : T,
       neighbours : NBL, bitmap : BM >
        < Random : RandomNGen | seed : M >
      =>
        < O : WSNode | backoffTimer : (if O' createsClosestCrossing
                                                    O (NBL (O' starting (D >= 0)))
                                       then setTC1(O, NBL (O' starting (D >= 0)), M)
                                       else T
                                       fi),
                         neighbours : NBL (O' starting (D >= 0)),
                      bitmap : updateBitmap(O, BM, O') >
        < Random : RandomNGen | seed : random(M) >

      if S =/= on /\ P > 0 /\
 not sensingAreaCovered(O, updateBitmap(O, BM, O'), O') /\
 existsUncoveredCrossings(O, NBL (O' starting (D >= 0))) .




    ***( Update timer to TC2 if all crossings are covered and
    ***  O' is the closest starting neighbour )***
    crl [recPowerOnWithStartingNeighbours] :
(PowerOnMsgFrom O' to O withDirection D)
< O : WSNode | remainingPower : P, status : S, backoffTimer : T,
       neighbours : NBL, bitmap : BM >
        < Random : RandomNGen | seed : M >
      =>
        < O : WSNode | backoffTimer : (if O' == findClosestStartingNeighbour(O,
    NBL (O' starting (D >= 0)))
                                       then setTC2(O, O', D,
      NBL (O' starting (D >= 0)), M)
                                       else T
                                       fi),
                         neighbours : NBL (O' starting (D >= 0)),
                 bitmap : updateBitmap(O, BM, O') >
        < Random : RandomNGen | seed : random(M) >

      if S =/= on /\ P > 0 /\
```

```
 existsStartingNeighbour(NBL (O' starting (D >= 0))) /\
 not sensingAreaCovered(O, updateBitmap(O, BM, O'), O') /\
 not existsUncoveredCrossings(O, NBL (O' starting (D >= 0))) .




    ***( Update timer to TC if all crossings are covered and
    ***  there's no starting neighbour and
    ***  O' is the closest neighbour )***
    crl [recPowerOnWithNeighbours] :
(PowerOnMsgFrom O' to O withDirection D)
< O : WSNode | remainingPower : P, status : S, backoffTimer : T,
       neighbours : NBL, bitmap : BM >
      =>
        < O : WSNode | backoffTimer : (if O' == findClosestNeighbour(O,
  NBL (O' starting (D >= 0)))
                                       then tc
                                       else T
                                       fi),
                       neighbours : NBL (O' starting (D >= 0)),
                   bitmap : updateBitmap(O, BM, O') >

      if S =/= on /\ P > 0 /\
 not existsStartingNeighbour(NBL (O' starting (D >= 0))) /\
 not sensingAreaCovered(O, updateBitmap(O, BM, O'), O') /\
 not existsUncoveredCrossings(O, NBL (O' starting (D >= 0))) .




    ***( Discard/ignore power-on messages received after powered on )***
    rl [discard] :
(PowerOnMsgFrom O' to O withDirection D)
< O : WSNode | status : on >
      =>
        < O : WSNode | > .

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




    ***( Restart the node when the round is over )***
    crl [restart] :
```

```
        < O : WSNode | roundTimer : 0, remainingPower : P >
      =>
        < O : WSNode | status : undecided,
                            neighbours : nil,
   bitmap : initBitmap(O),
   hasVolunteered : undecided,
   backoffTimer : INF,
   roundTimer : roundTime,
   volunteerProb : 1000 / n >
     if P > 0 .


endtom)







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


(tomod OGDC-RTM is
    protecting OGDC .


    var O  : Oid .
    vars T T' : Time .
    vars TI TI' : TimeInf .
    var P : Nat .
    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 >) = INF .

    ceq 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)
       if P > 0 .

    ceq 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)
       if P > 0 .


    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 .



    eq sensingAreaSize = 5000 .

    eq n = 100 .


    msg workingNodes      : Nat Int -> Msg .

    op numWorkingNodes     : Configuration -> Nat [frozen (1)] .
    op countNodes          : Configuration -> Nat [frozen (1)] .


    msg coveragePercentage : Nat Int -> Msg .

    op coveragePercentage  : Configuration -> Nat [frozen (1)] .
    op calcCoveragePercentage  : Bitmap Nat Nat -> Nat .


    msg totalRemainingPower  : Nat Nat -> Msg .

    op calcTotalRemainingPower : Configuration -> Nat [frozen (1)] .




    op genNodes : Nat Nat -> Configuration . *** seed numNodes
    op genNodes : Nat Nat Nat -> Configuration .
```

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


op sensingArea : -> Bitmap .
op makeAreaColumns : Nat Nat -> Bitmap .
op makeAreaRows : Nat -> BitList .




op updateArea : Bitmap Configuration -> Bitmap .
op steadyStatePhase : Configuration -> Bool [frozen (1)] .
op extractActiveNodes : Configuration -> Configuration [frozen (1)] .
op extractOid : Configuration -> OidSet .


op sensor-area-covered : -> Prop [ctor] .
op steady-state-phase : -> Prop [ctor] .






var F : Float .
vars M M' N N' P : Nat .
var D : Int .
var T : Time .
var BIT : Bit .
var BITL : BitList .
var BM : Bitmap .
vars OS OS' : OidSet .
vars O O' O'' : Oid .
var L : Location .
vars S S' : Status .
var NB : Neighbour .
var NBL : NeighbourList .
vars C C' C'' : Configuration .
var NEC : NEConfiguration .
var OC : ObjectConfiguration .
var MSG : Msg .
```

```
    eq { workingNodes(N, 0) C } = { dly(workingNodes(N, numWorkingNodes(C)),INF)
 dly(workingNodes(N + 1, 0), roundTime) C } .


    eq { coveragePercentage(N, 0) C } = { dly(coveragePercentage(N,
coveragePercentage(C)), INF)
       dly(coveragePercentage(N + 1, 0),
roundTime) C } .


    eq { totalRemainingPower(N, 0) C } = { dly(totalRemainingPower(N,
   calcTotalRemainingPower(C)), INF)
        dly(totalRemainingPower(N + 1, 0),
   roundTime) C } .
```

```
    eq genNodes(M, M') = genNodes(M, M', M') .

    ceq genNodes(M, s(0), M') = < Random : RandomNGen | seed : repeatRandom(M, 3) >
   < L : WSNode |
              remainingPower : lifetime, status : undecided,
                              neighbours : nil, bitmap : initBitmap(L),
                              backoffTimer : INF, roundTimer : roundTime,
                              volunteerProb : 1000 / M',
      volunteeredWithSuccess : undecided >
if L := random(M) rem sensorAreaSize - (sensorAreaSize / 2) .
                random(random(M)) rem sensorAreaSize - (sensorAreaSize / 2) .

    ceq genNodes(M, s(s(N)), M') =
     < L : WSNode |
                remainingPower : lifetime, status : undecided,
                                neighbours : nil, bitmap : initBitmap(L),
                                backoffTimer : INF, roundTimer : roundTime,
                                volunteerProb : 1000 / M',
        volunteeredWithSuccess : undecided >
    genNodes(repeatRandom(M, 3), s(N), M')
if L := random(M) rem sensorAreaSize - (sensorAreaSize / 2) .
                random(random(M)) rem sensorAreaSize - (sensorAreaSize / 2) .
```

```
    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 .
```

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

```
    eq numWorkingNodes(C) = countNodes(extractActiveNodes(C)) .
    eq countNodes(none) = 0 .
    eq countNodes(< O : WSNode | > C) = 1 + countNodes(C) .
```

```
    eq coveragePercentage(C) = calcCoveragePercentage(updateArea(sensorArea, C), 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 | > C) = calcTotalRemainingPower(C) .
    eq calcTotalRemainingPower(MSG C) = calcTotalRemainingPower(C) .
    eq calcTotalRemainingPower(< O : WSNode | remainingPower : P > C) =
   P + calcTotalRemainingPower(C) .



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



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

    eq extractOid(none) = none .
    eq extractOid(< O : WSNode | > C ) = O ; extractOid(C) .



    eq { C } |= sensor-area-covered = checkCoverage(updateArea(sensorArea, C)) .

    eq { C } |= steady-state-phase = steadyStatePhase(C) .
```

```
endtom)

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



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



****************************************************
*****         TEST REWRITES AND SEARCHES       *****
****************************************************


**** WORKING NODES AND COVERAGE VS DEPLOYED NODES
***(tfrew {genNodes(1, 1000) dly(workingNodes(0,0),roundTime - 1)
           dly(coveragePercentage(0,0),roundTime - 1)} in time <= roundTime .)

**** COVERAGE AND POWER VS TIME
***(tfrew {genNodes(9569, 75) dly(coveragePercentage(0, 0),roundTime - 1)
           dly(totalRemainingPower(0, 0),roundTime - 1)} in time <= roundTime * 50 .)
***(tfrew {genNodes(1, 48) dly(coveragePercentage(0, 0),roundTime - 1)
           dly(totalRemainingPower(0, 0),roundTime - 1)} in time <= roundTime * 50 .)

**** ALPHA LIFETIME VS ALPHA
***(tfrew {genNodes(47, 75) dly(coverage(50),INF) } in time <= roundTime * 50 .)
***(tfrew {genNodes(1, 48) dly(coverage(50),INF) } in time <= roundTime * 50 .)



**** ALPHA LIFETIME VS DEPLOYED NODES
***(tfrew {genNodes(1, 64) dly(coverage(98),INF) } in time <= roundTime * 200 .)
***(tfrew {genNodes(45, 80) dly(coverage(98),INF) } in time <= roundTime * 200 .)
***(tfrew {genNodes(35, 20) dly(coverage(98),INF) } in time <= roundTime * 200 .)
***(tfrew {genNodes(8921, 125) dly(coverage(98),INF) } in time <= roundTime * 200 .)




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

```
***(find earliest {genNodes(1, 5)} =>* { C:Configuration } such that
 steadyStatePhase(C:Configuration) .)

***(mc {genNodes(1,5)} |=t (steady-state-phase => [] steady-state-phase)
         in time < roundTime .)


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

***(tsearch [1] {genNodes(5,5)} =>* { < O:Oid : WSNode | status : on,
      bitmap : BM:Bitmap,
      ATTS:AttributeSet > C:Configuration }
    such that checkCoverage(BM:Bitmap) *** sensor area covered
    in time <= roundTime .) *** 619

***q
```