Modelling Clock Synchronization in the Chess gMAC WSN Protocol*

Mathijs Schuts Feng Zhu Faranak Heidarian†
Frits Vaandrager
Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
M.Schuts@student.ru.nl, FengZhu@student.ru.nl, faranak@cs.ru.nl, F.Vaandrager@cs.ru.nl

We present a detailled timed automata model of the clock synchronization algorithm that is currently being used in a wireless sensor network (WSN) that has been developed by the Dutch company Chess. Using the UPPAAL model checker, we establish that in certain cases a static, fully synchronized network may eventually become unsynchronized if the current algorithm is used, even in a setting with infinitesimal clock drifts.

1 Introduction

Wireless sensor networks consist of autonomous devices that communicate via radio and use sensors to cooperatively monitor physical or environmental conditions. In this paper, we formally model and analyze a distributed algorithm for clock synchronization in wireless sensor networks that has been developed by the Dutch company Chess in the context of the MyriaNed project [15]. Figure 1 displays a sensor node developed by Chess. The algorithm that we consider is part of the *Medium Access Control*

Figure 1: Chess MyriaNode 2.4 Ghz wireless sensor node

(MAC) layer, which is responsible for the access to the wireless shared channel. Within its so-called gMAC protocol, Chess uses a Time Division Multiple Access (TDMA) protocol. Time is divided in fixed length *frames*, and each frame is subdivided into *slots* (see Figure 2). Slots can be either *active* or *sleeping* (*idle*). During active slots, a node is either listening for incoming messages from neighboring nodes (*RX*) or it is sending a message (*TX*). During sleeping slots a node is switched to energy saving mode. Since energy efficiency is a major concern in the design of wireless sensor networks, the number

*Research supported by the European Community's Seventh Framework Programme under grant agreement no 214755 (QUASIMODO) and by the DFG/NWO bilateral cooperation project ROCKS.
†Research supported by NWO/EW project 612.064.610 Abstraction Refinement for Timed Systems (ARTS).

S. Andova et.al. (Eds.): Workshop on Quantitative Formal Methods:
Theory and Applications (QFM'09)
EPTCS 13, 2009, pp. 41–54, doi:10.4204/EPTCS.13.4

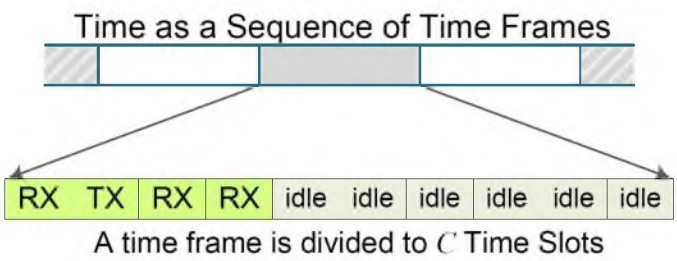

Figure 2: The structure of a time frame

of active slots is typically much smaller than the total number of slots (less than 1% in the current implementation). The active slots are placed in one contiguous sequence which currently is placed at the beginning of the frame. A node can only transmit a single message per time frame, during its TX slot. The protocol takes care that neighboring nodes have different TX slots.

One of the greatest challenges in the design of the MAC layer is to find suitable mechanisms for clock synchronization: we must ensure that whenever some node is sending all its neighbors are listening. Sensor nodes come equipped with a crystal clock, which may drift. This may cause the TDMA time slot boundaries to drift and thus lead to situations in which nodes get out of sync. To overcome this problem nodes will have to adjust their clocks now and then. Also, the notion of *guard time* is introduced: at the beginning of its TX slot, a sender waits a certain amount of time to ensure that all its neighbors are ready to receive messages. Similarly, a sender does not transmit for a certain amount of time at the end of its TX slot. In order to save energy it is important to reduce these guard times to a minimum. Many clock synchronization protocols have been proposed for wireless sensor networks, see e.g. [16, 5, 17, 12, 1, 11, 14]. However, these protocols (with the exception of [17, 1] and possibly [14]) involve a computation and/or communication overhead that is unacceptable given the extremely limited resources (energy, memory, clock cycles) available within the Chess nodes.

To experiment with its designs, Chess currently builds prototypes and uses advanced simulation tools. However, due to the huge number of possible network topologies and clock speeds of nodes, it is difficult to discover flaws in the clock synchronization algorithm via these methods. Timed automata model checking has been succesfully used for the analysis of worst case scenarios for protocols that involve clock synchronization, see for instance [4, 8, 19]. To enable model checking, models need to be much more abstract than for simulation, and also the size of networks that can be tackled is much smaller, but the big advantage is that the full state space of the model can be explored.

In this paper, we present a detailed model of the Chess gMAC algorithm using the input language of the timed automata model checking tool UPPAAL [3]. Another UPPAAL model for the gMAC algorithm is presented in [9], but that model deviates and abstracts from several aspects in the implementation in order to make verification feasible. The aim of the present paper is to construct a model that comes as close as possible to the specification of the clock synchronization algorithm presented in [15]. Nevertheless, our model still does not incorporate some features of the full algorithm and network, such as dynamic slot allocation, synchronization messages, uncertain communication delays, and unreliable radio communication. At places where the informal specification of [15] was incomplete or ambiguous, the engineers from Chess kindly provided us with additional information on the way these issues are resolved in the current implementation of the network [20]. In the current implementation of Chess, a node can only adjust its clock once every time frame during the sleeping period, using an extension of

the Median algorithm of [17]. This contrasts with the approach in [9] in which a sensor node may adjust its clock after every received message. In the present paper we faithfully model the Median algorithm as implemented by Chess. Another feature of the gMAC algorithm that was not addressed in [9] but that we model in this paper is the radio switching time: there is some time involved in the transition from sending mode to receiving mode (and vice versa), which in some cases may affect the correctness of the algorithm.

The Median algorithm works reasonably well in practice, but by means of simulation experiments, Assegei [1] already exposed some flaws in the algorithm: in some test cases where new nodes join or networks merge, the algorithm fails to converge or nodes may stay out of sync for a certain period of time. Our analysis with UPPAAL confirms these results. In fact, we show that the situation is even worse: in certain cases a static, fully synchronized network may eventually become unsynchronized if the Median algorithm is used, even in a setting with infinitesimal clock drifts.

In Section 2, we explain the gMAC algorithm in more detail. Section 3 describes our UPPAAL model of gMAC. In Section 4, the analysis results are described. Finally, in Section 5, we draw some conclusions. In this paper, we assume that the reader has some basic knowledge of the timed automaton tool UPPAAL. For a detailed account of UPPAAL, we refer to [3, 2] and to `http://www.uppaal.com`. The UPPAAL model described in this paper is available at `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/chess09/`.

2 The gMAC Protocol

In this section we provide additional details about the gMAC protocol as it has currently been implemented by Chess.

2.1 The Synchronization Algorithm

In each frame, each node broadcasts one message to its neighbors. The timing of this message is used for synchronization purposes: a receiver may estimate the clock value of a sender based on the time when the message is received. Thus there is no need to send around (logical) clock values. In the current implementation of Chess, clock synchronization is performed once per frame using the following algorithm [1, 20]:

1. In its sending slot, a node broadcasts a packet which contains its transmission slot number.
2. Whenever a node receives a message it computes the `phase error`, that is the difference (number of clock cycles) between the expected receiving time and the actual receiving time of the incoming message. Note that the difference between the sender's sending slot number (which is also the current slot number of the sender) and the current slot number of the receiving node must also be taken into account when calculating the phase errors.
3. After the last active slot of each frame, a node calculates the `offset` from the phase errors of all incoming messages in this frame with the following algorithm:

```
if (number of received messages == 0)
        offset = 0;
else if (number of received messages <= 2)
        offset = the phase error of the first received message * gain;
else
        offset = the median of all phase errors * gain
```

Here `gain` is a coefficient with value 0.5, used to prevent oscillation of the clock adjustment.

4. During the sleeping period, the local clock of each node is adjusted by the computed `offset` obtained from step 3.

In situations when two networks join, it is possible that the phases of these networks differ so much that the nodes in one network are in active slots whereas the nodes in the other network are in sleeping slots and vice versa. In this case, no messages can be exchanged between two networks. Therefore in the Chess design, a node will send an extra message in one (randomly selected) sleeping slot to increase the chance that networks can communicate and synchronize with each other. This slot is called the synchronization slot and the message is in the same format as in the transmission slot. The extreme value of `offset` can be obtained when two networks join: it may occur that the `offset` is larger than half the total number of clock cycles of sleeping slots in a frame. Chess uses another algorithm called `join` to handle this extreme case. We decided not to model joining of networks and synchronization messages because currently we do not have enough information about the `join` algorithm.

2.2 Guard Time

The correctness condition for gMAC that we would like to establish is that whenever a node is sending all its neighbors are in receiving mode. However, at the moment when a node enters its TX slot we cannot guarantee, due to the phase errors, that its neighbors have entered the corresponding RX slot. This problem is illustrated in Figure 3 (a). Given two nodes 1 and 2, if a message is transmitted during the entire sending slot of node 1 then this message may not be successfully received by node 2 because of the imperfect slot alignment. Taking the clock of node 1 as a reference, the clock of node 2 may drift backwards or forwards. In this situation, node 1 and node 2 may have a different view of the current slot number within the time interval where node 1 is sending a message.

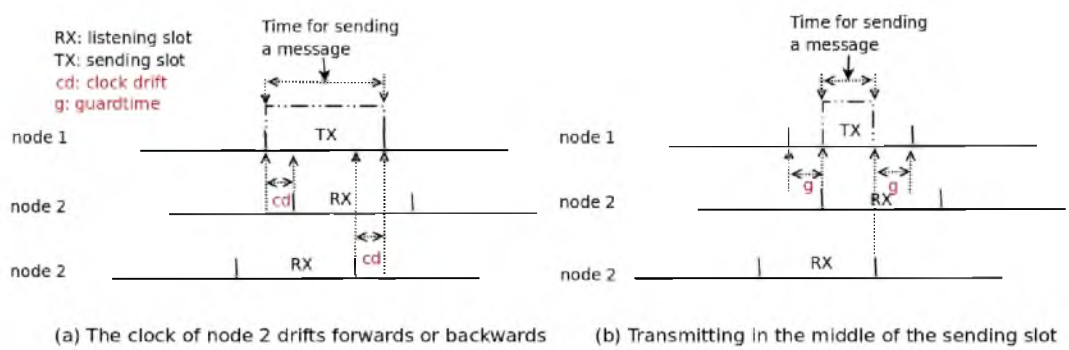

Figure 3: The need for introducing guard times

To cope with this problem, messages are not transmitted during the entire sending slot but only in the middle, as illustrated in Figure 3 (b). Both at the beginning and at the end of its sending slot, node 1 does not transmit for a preset period of g clock ticks, in order to accomodate the forwards and backwards clock drift of node 2. Therefore, the time available for transmission equals the total length of the slot minus 2g clock ticks.

2.3 Radio Switching Time

The radio of a wireless sensor node can either be in sending mode, or in receiving mode, or in idle mode. Switching from one mode to another takes time. In the current implementation of the Chess gMAC protocol, the radio switching time is around 130μsec. The time between clock ticks is around 30μsec and the guard time g is 9 clock ticks. Hence, in the current implementation the radio switching time is smaller than the guard time, but this may change in future implementations. If the first slot in a frame is an RX slot, then the radio is switched to receiving mode some time before the start of the frame to ensure that the radio will receive during the full first slot. However if there is an RX slot after the TX slot then, in order to keep the implementation simple, the radio is switched to the receiving mode only at the start of the RX slot. Therefore messages arriving in such receiving slots may not be fully received. This issue may also affect the performance of the synchronization algorithm.

3 Uppaal Model

In this section, we describe the UPPAAL model that we constructed of the gMAC protocol.

We assume a finite, fixed set of wireless nodes $\mathsf{Nodes} = \{0, \ldots, \mathsf{N}-1\}$. The behavior of an individual node $\mathsf{id} \in \mathsf{Nodes}$ is described by five timed automata Clock(id), Receiver(id), Sender(id), Synchronizer(id) and Controller(id). Figure 4 shows how these automata are interrelated. All components interact with the clock, although this is not shown in Figure 4. Automaton Clock(id) models the hardware clock of node id, automaton Sender(id) the sending of messages by the radio, automaton Receiver(id) the receiving part of the radio, automaton Synchronizer(id) the synchronization of the hardware clock, and automaton Controller(id) the control of the radio and the clock synchronization.

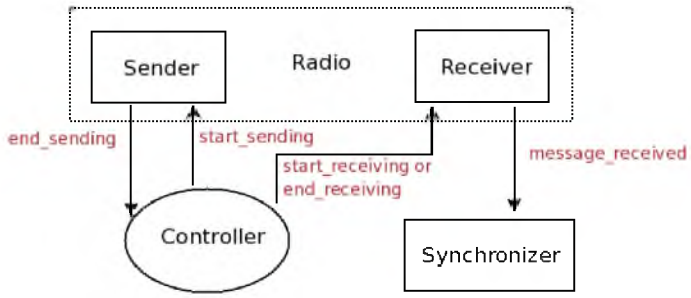

Figure 4: Message flow in the model

Table 1 lists the parameters that are used in the model (constants in UPPAAL terminology), together with some basic constraints. The domain of all parameters is the set of natural numbers. We will now describe the five automaton templates used in our model.

Clock Timed automaton Clock(id) models the behavior of the hardware clock of node id. The automaton is shown in Figure 5. At the start of the system state variable csn[id], that records the current slot number, is initialized to $\mathsf{C}-1$, that is, to the last sleeping slot. Hardware clocks are not perfect and so we assume a minimal time min[id] and a maximal time max[id] between successive clock ticks. Integer variable clk[id] records the current value of the hardware clock. For convenience (and to reduce the size of the state space), we assume that the hardware clock is reset at the end of each slot, that is after k_0

Parameter	Description	Constraints
N	number of nodes	$0 < \mathsf{N}$
C	number of slots in a time frame	$0 < \mathsf{C}$
n	number of active slots in a time frame	$0 < \mathsf{n} \leq \mathsf{C}$
tsn[id]	TX slot number for node id	$0 \leq \mathsf{tsn}[\mathsf{id}] < \mathsf{n}$
k_0	number of clock ticks in a time slot	$0 < k_0$
g	guard time	$0 < \mathsf{g}$
r	radio switch time	$0 \leq \mathsf{r}$
min[id]	minimal time between two clock ticks of node id	$0 < \mathsf{min}[\mathsf{id}]$
max[id]	maximal time between two clock ticks of node id	$\mathsf{min}[\mathsf{id}] \leq \mathsf{max}[\mathsf{id}]$

Table 1: Protocol parameters

clock ticks. Also, a state variable csn[id], which records the current slot number of node id, is updated each time at the start of a new slot.

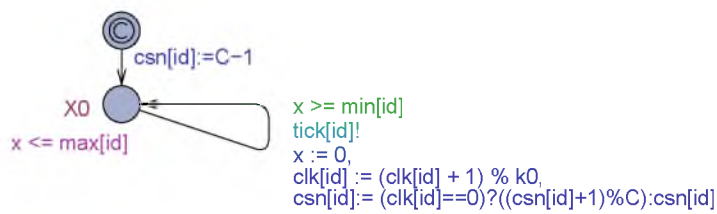

Figure 5: Automaton Clock[id]

Sender The sending behavior of the radio is described by the automaton Sender[id] shown in Figure 6. The behavior is rather simple. When the controller asks the sender to transmit a message (via a

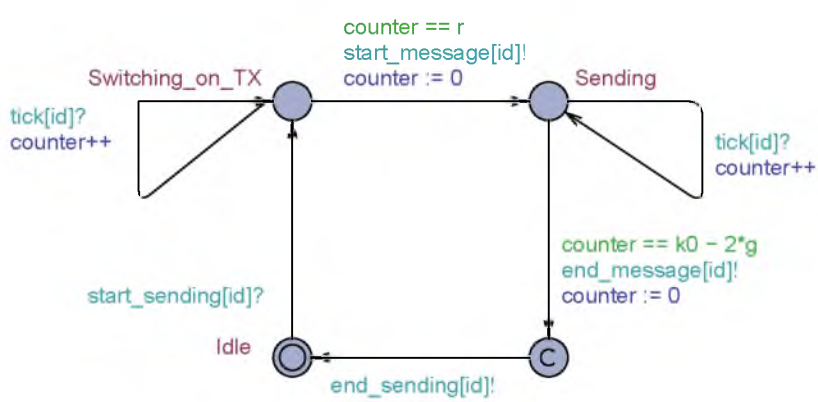

Figure 6: Automaton Sender[id]

start_sending[id] signal), the radio first switches to sending mode (this takes r clock ticks) and then transmits the message (this takes $k_0 - 2 \cdot g$ ticks). Immediately after the message transmission has been completed, an end_sending[id] signal is sent to the controller to indicate that the message has been sent.

Receiver The automaton Receiver[id] models the receiving behavior of the radio. The automaton is shown in Figure 7. Again the behavior is rather simple. When the controller asks the receiver to start receiving, the receiver first switches to receiving mode (this takes r ticks). After that, the receiver may receive messages from all its neighbors. A function neighbor is used to encode the topology of the network: neighbor(j, id) holds if messages sent by j can be received by id. Whenever the receiver detects the end of a message transmission by one of its neighbors, it immediately informs the synchronizer via a message_received[id] signal. At any moment, the controller can switch off the receiver via an end_receiving[id] signal.

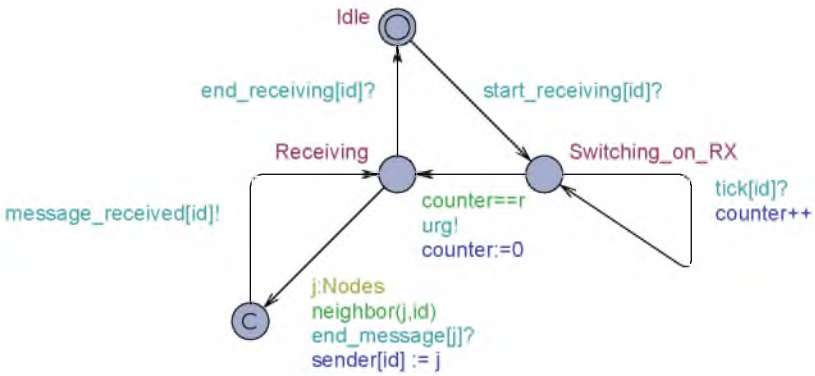

Figure 7: Automaton Receiver[id]

Controller The task of the Controller[id] automaton, displayed in Figure 8, is to put the radio in sending and receiving mode at the appropriate moments. Figure 9 shows the definition of the predicates used in this automaton. The radio should be put in sending mode r ticks before message transmission starts (at time g in the transmission slot of id). If $r > g$ then the sender needs to be activated $r - g$ ticks before the end of the slot that precedes the transmission slot. Otherwise, the sender must be activated at tick $g - r$ of the transmission slot. If the first slot in a frame is an RX slot, then the radio is switched to receiving mode r time units before the start of the frame to ensure that the radio will receive during the full first slot. However if there is an RX slot after the TX slot then, as described in Section 2.3, the radio is switched to the receiving mode only at the start of the RX slot. The controller stops the radio receiver whenever either the last active slot has passed or the sender needs to be switched on.

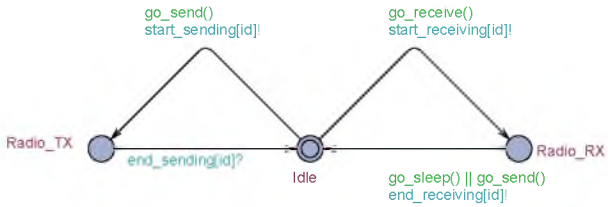

Figure 8: Automaton Controller[id]

```
bool go_send(){return (r>g)
 ?((csn[id]+1)%C==tsn[id] && clk[id]==k0-(r-g))
 :(csn[id]==tsn[id] && clk[id]==g-r);}

bool go_receive(){return
(r>0 && 0!=tsn[id] && csn[id]==C-1 && clk[id]==k0-r)
|| (r==0 && 0!=tsn[id] && csn[id]==0)
|| (0<csn[id] && csn[id]<n && csn[id]-1==tsn[id]);}

bool go_sleep(){return csn[id]==n;}
```

Figure 9: Predicates used in Controller[id]

All the channels used in the Controller[id] automaton (start_sending, end_sending, start_receiving, end_receiving and synchronize) are urgent, which means that these signals are sent at the moment when the transitions are enabled.

Synchronizer Finally, automaton Synchronizer[id] is shown in Figure 10. The automaton maintains a list of phase differences of all messages received in the current frame, using a local array phase_errors. Local variable msg_counter records the number of messages received. Whenever the receiver gets a mes-

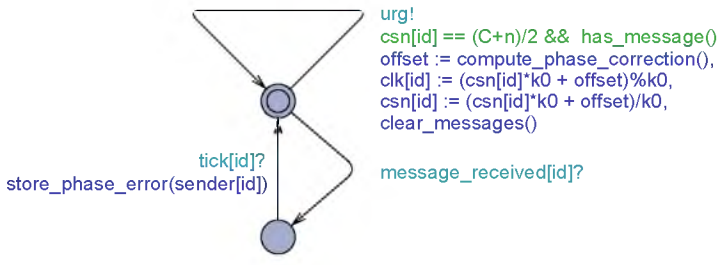

Figure 10: Automaton Synchronizer[id]

```
void store_phase_error(int sender)
{
phase_errors[msg_counter] =
   (tsn[sender] * k0 + k0 - g)
        - (csn[id] * k0 + clk[id]);
msg_counter++
}
```

Figure 11: Function used in Synchronizer[id]

sage from a neighboring node (message_received[id]), the synchronizer computes and stores the phase difference using the function store_phase_error at the next clock tick. Here the phase difference is defined as the expected time at which the message transmission ends (`tsn[sender] * k0 + k0 - g`) minus the actual time at which the message transmission ends (`csn[id] * k0 + clk[id]`), counting from the start of the frame. The complete definition is listed in Figure 11. Recall that in our model we abstract from transmission delays.

As explained in Section 2.1, the synchronizer computes the value of the phase correction (offset) and adjusts the clock during the sleeping period of a frame.[1] Hence, in order to decide in which slot we may perform the synchronization, we need to know the maximal phase difference between two nodes. In our model, we assume no joining of networks. When a node receives a message from another node, the phase difference computed using this message will not exceed the length of an active period. Otherwise one of these two nodes will be in sleeping period while the other is sending, hence no message can be received at all. In practice, the number of sleeping slots is much larger than the number of active slots. Therefore it is safe to perform the adjustment in the middle of sleeping period because the desired property described above holds. When the value of `gain` is smaller than 1 the maximal phase difference will be even smaller.

The function of compute_phase_correction implements exactly the algorithm listed in Section 2.1.

4 Analysis Results

In this section, we present some verification results that we obtained for simple instances of the model that we described in Section 3. We checked the following invariant properties using the UPPAAL model checker:

```
INV1 : A[] forall (i: Nodes) forall (j : Nodes)
   SENDER(i).Sending && neighbor(i,j)imply RECEIVER(j).Receiving
```

[1] Actually, in the implementation the offset is used to compute the corrected *wakeup time*, that is the moment when the next frame will start [20]. In our model we reset the clock, but this should be equivalent.

```
INV2 : A[] forall (i:Nodes) forall (j:Nodes) forall (k:Nodes)
   SENDER(i).Sending && neighbor(i,k) && SENDER(j).Sending && neighbor(j,k)
                    imply i == j

INV3 : A[] not deadlock
```

The first property states that always when some node is sending, all its neighbors are listening. The second property states that never two different neighbors of a given node are sending simultaneously. The third property states that the model contains no deadlock, in the sense that in each reachable state at least one component can make progress. The three invariants are basic sanity properties of the gMAC protocol, at least in a setting with a static topology and no transmission failures.

We used UPPAAL on a Sun Fire X4440 machine (with 4 Opteron 8356 2.3 Ghz quad-core processors and 128 Gb DDR2-667 memory) to verify instances of our model with different number of nodes, different network topologies and different parameter values. Table 2 lists some of our verification results, including the resources UPPAAL needed to verify if the network is synchronized or not. In all experiments, $\mathsf{C} = 10$ and $\mathsf{k}_0 = 29$.

Clearly, the values of network parameters, in particular clock parameters min and max, affect the result of the verification. Table 2 shows several instances where the protocol is correct for perfect clocks ($\mathsf{min} = \mathsf{max}$) but fails when we decrease the ratio $\frac{\mathsf{min}}{\mathsf{max}}$. It is easy to see that the protocol will always fail when $\mathsf{r} \geq \mathsf{g}$. Consider any node i that is not the last one to transmit within a frame. Right after its sending slot, node i needs r ticks to get its radio into receiving mode. This means that — even with perfect clocks — after g ticks another node already has started sending even though the radio of node i is not yet receiving. Even when $\mathsf{r} < \mathsf{g}$, the radio switching time has a clear impact on correctness: the larger the radio switching time is, the larger the guard time has to be in order to ensure correctness. Using UPPAAL, we can fully analyze line topologies with at most seven nodes if all clocks are perfect. For larger networks UPPAAL runs out of memory. A full parametric analysis of this protocol will be challenging, also due to the impact of the network topology and the selected slot allocation. Using UPPAAL, we discovered that for certain topologies and slot allocations the Median algorithm may always violate the above correctness assertions, irrespective of the choice of the guard time. For example, in a 4 node-network with clique topology and min and max of 100.000 and 100.001, respectively, if the median of the clock drifts of a node becomes -1, the median algorithm divides it by 2 and generates 0 for clock correction value and indeed no synchronization happens. If this scenario repeats in three consecutive time frames for the same node, that node runs $\mathsf{g} = 3$ clock cycles behind and gets out of sync.

Another example in which the algorithm may fail is displayed in Figure 12. This network has 4 nodes, connected by a line topology, that send in slots 1, 2, 3, 1, respectively. Since all nodes have at

Figure 12: A problematic network configuration

most two neighbors, the Median algorithm prescribes that nodes will correct their clocks based on the first phase error that they see in each frame. For the specific topology and slot allocation of Figure 12, this means that node 0 adjusts its clock based on phase errors of messages it gets from node 1, node 1 adjusts

N/n	Topology	g	r	$\frac{\min}{\max}$	CPU Time	Peak Memory Usage	Sync
3/3	clique	2	0	1	1.944 s	24,180 KB	YES
3/3	clique	2	0	$\frac{100,000}{100,001}$	492.533 s	158,064 KB	NO
3/3	clique	2	1	1	1.976 s	68.144 KB	YES
3/3	clique	2	0	$\frac{100,000}{100,001}$	116.68 s	68,144 KB	NO
3/3	line	2	0	1	1.068 s	68,144 KB	YES
3/3	line	2	0	$\frac{100,000}{100,000}$	441.308 s	68,144 KB	NO
3/3	line	2	1	1	1.041 s	68,144 KB	YES
3/3	line	2	1	$\frac{100,000}{100,000}$	99.274 s	68,144 KB	NO
3/3	clique	3	0	1	1.851 s	28,040 KB	YES
3/3	clique	3	0	$\frac{100,000}{100,001}$	575.085 s	272,312 KB	NO
3/3	clique	4	0	$\frac{350}{351}$	115.166 s	516,636 KB	NO
3/3	clique	4	0	$\frac{351}{352}$	147.864 s	630,044 KB	YES
3/3	clique	3	2	1	1.827 s	24,184 KB	YES
3/3	clique	3	2	$\frac{100,000}{100,001}$	109.633 s	26,056 KB	NO
3/3	clique	4	2	$\frac{100,000}{100,001}$	533.345 s	350,504 KB	NO
3/3	clique	5	2	$\frac{587}{588}$	72.473 s	332,552 KB	NO
3/3	clique	5	2	$\frac{588}{589}$	99.101 s	407,884 KB	YES
3/3	clique	3	5	1	0.076 s	21,884 KB	NO
3/3	line	3	0	1	1.05 s	23,348 KB	YES
3/3	line	3	0	$\frac{451}{452}$	29.545 s	148,012 KB	NO
3/3	line	3	0	$\frac{452}{453}$	35.257 s	148,012 KB	YES
3/3	line	3	2	1	1.052 s	22,916 KB	YES
3/3	line	3	2	$\frac{100,000}{100,001}$	82.383 s	78,360 KB	NO
3/3	line	4	2	$\frac{100,000}{100,001}$	414.201 s	53,752 KB	NO
3/3	line	5	2	$\frac{453}{454}$	33.16 s	147,796 KB	NO
3/3	line	5	2	$\frac{454}{455}$	38.811 s	162,184 KB	YES
3/3	line	3	5	1	0.048 s	78,360 KB	NO
4/4	clique	3	0	1	231.297 s	1,437,643 KB	YES
4/4	clique	3	0	$\frac{450}{451}$	Memory Exhausted		
4/4	clique	3	2	1	229.469 s	1,438,368 KB	YES
4/4	clique	3	2	$\frac{100,000}{100,001}$	14,604.531 s	2,317,040 KB	NO
4/3	line	3	0	1	4.749s s	94,748 KB	YES
4/3	line	3	0	$\frac{450}{451}$	Memory Exhausted		
4/3	line	3	2	1	4.738 s	94,748 KB	YES
4/3	line	3	2	$\frac{100,000}{100,001}$	1,923.655 s	1,264,844 KB	YES
5/5	clique	3	0	1	Memory Exhausted		
5/5	clique	3	2	1	Memory Exhausted		
5/3	line	3	0	1	46.54 s	249,976 KB	YES
5/3	line	3	2	1	46.489 s	250,880 KB	YES
6/3	line	3	0	1	508.19 s	2,316,416 KB	YES
6/3	line	3	2	1	502.871 s	2,317,040 KB	YES
7/3	line	3	0	1	Memory Exhausted		
7/3	line	3	2	1	Memory Exhausted		

Table 2: Model checking experiments

its clock based on messages from node 0, node 2 adjusts its clock based on messages from node 3, and node 3 adjusts its clock based on messages from node 2. Hence, for the purpose of clock synchronization, we have two disconnected networks! Thus, if the clock rates of nodes 0 and 1 are lower than the clock rates of nodes 2 and 3 by just an arbitrary small margin, then two subnetworks will eventually get out of sync. These observations are consistent with results that we obtained using UPPAAL. If, for instance, we set $\mathsf{min}[\mathsf{id}] = 99$ and $\mathsf{max}[\mathsf{id}] = 100$, for all nodes id then neither `INV1` nor `INV2` holds. In practice, it is unlikely that the above scenario will occur due to the fact that in the implementation slot allocation is random and dynamic. Due to regular changes of the slot allocation, with high probability node 1 and node 2 will now and then adjusts their clocks based on messages they receive from each other.

However, variations of the above scenario may occur in practice, even in a setting with dynamic slot allocation. In fact, the above synchronization problem is also not restricted to line topologies. We call a subset C of nodes in a network a *community* if each node in C has more neighbors within C than outside C [13]. For *any* network in which two disjoint communities can be identified, the Median algorithm allows for scenarios in which these two parts become unsynchronized. Due to the median voting mechanism, the phase errors of nodes outside a community will not affect the nodes within this community, independent of the slot allocation. Therefore, if nodes in one community A run slow and nodes in another community B run fast then the network will become unsynchronized eventually, even in a setting with infinitesimal clock drifts. Figure 13 gives an example of a network with two communities.

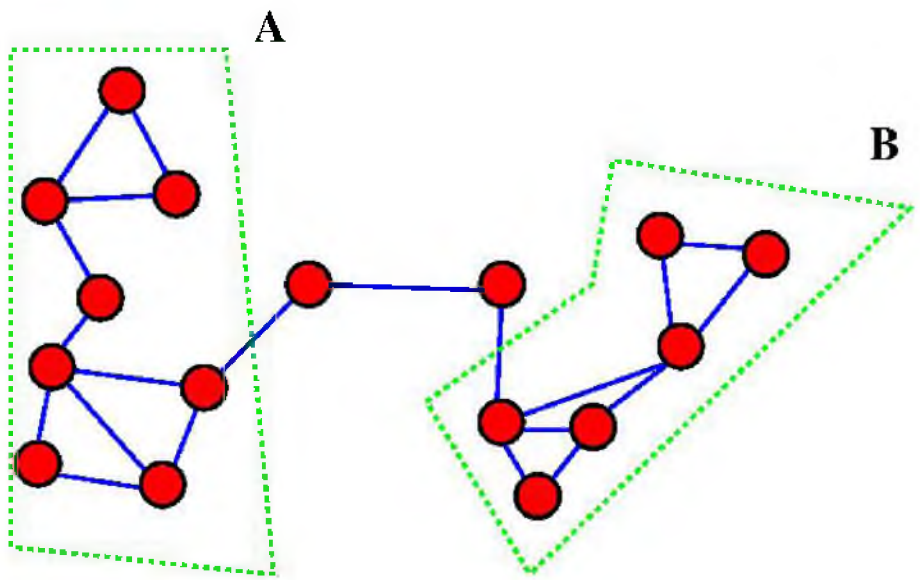

Figure 13: Another problematic network configuration with two communities

Using UPPAAL, we succeeded to analyze instances of the simple network with two communities displayed in Figure 14. The numbers on the vertices are the node identifiers and the transmission slot numbers, respectively. Table 3 summarizes the results of our model checking experiments.

We still need to explore how realistic our counterexamples are. We believe that network topologies with multiple communities occur in many WSN applications. Nevertheless, in practice the gMAC protocol appears to perform quite well for static networks. It might be that problems do not occur so often in practice due to the probabilistic distributions of clock drift and jitter.

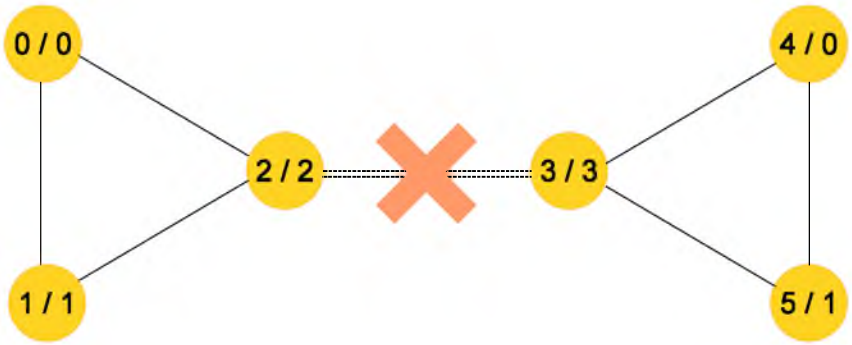

Figure 14: A network with two communities that we analyzed using UPPAAL

g	r	Fast Clock Cycle Length	Slow Clock Cycle Length	CPU Time	Peak Memory Usage
2	0	1	1	Memory Exhausted	
2	0	99	100	457.917 s	2,404,956 KB
2	1	99	100	445.148 s	2,418,032 KB
3	0	99	100	416.796 s	2,302,548 KB
3	2	1	1	Memory Exhausted	
3	2	99	100	22.105 s	83,476 KB
3	2	451	452	798.121 s	3,859,104 KB
3	2	452	453	Memory Exhausted	
4	0	99	100	424.935 s	2,323,004 KB
4	1	99	100	464.503 s	2,462,176 KB
4	2	99	100	420.742 s	2,323,952 KB

Table 3: Model checking experiments of a network with two communities

5 Conclusions

We presented a detailled UPPAAL model of relevant parts of the clock synchronization algorithm that is currently being used in a wireless sensor network that has been developed by Chess [15, 20]. The final model that we presented here may look simple, but the road towards this model was long and we passed through numerous intermediate versions on the way. Using UPPAAL, we established that in certain cases a static, fully synchronized network may eventually become unsynchronized if the current Median algorithm is used, even in a setting with infinitesimal clock drifts.

In [9], we proposed a slight variation of the gMAC algorithm that does not have the correctness problems of the Median algorithm. However, our algorithm still has to be tested in practice. Assegei [1] proposed and simulated three alternative algorithms, to be used instead of the Median algorithm, in order to achieve decentralized, stable and energy-efficient synchronization of the Chess gMAC protocol. It should be easy to construct UPPAAL models for Assegei's algorithms: basically, we only have to modify the definition of the `compute_phase_correction` function. Recently, Pussente & Barbosa [14], also proposed a very interesting new clock synchronization algorithm — in a somewhat different setting — that achieves an $O(1)$ worst-case skew between the logical clocks of neighbors. Much additional

research is required to analyze correctness and performance of these algorithms in the realistic settings of Chess with large networks, message loss, and network topologies that change dynamically. Starting from our current UPPAAL model, it should be relatively easy to construct models for the alternative synchronization algorithms in order to explore their properties.

Analysis of clock synchronization algorithms for wireless sensor networks is an extremely challenging area for quantitative formal methods. One challenge is to come up with the right abstractions that will allow us to verify larger instances of our model. Another challenge is to make more detailled (probabilistic) models of radio communication and to apply probabilistic model checkers and specification tools such as PRISM [10] and CaVi [6].

Several other recent papers report on the application of UPPAAL for the analysis of protocols for wireless sensor networks, see e.g. [7, 6, 18, 9]. In [21], UPPAAL is also used to automatically test the power consumption of wireless sensor networks. Our paper confirms the conclusions of [7, 18]: despite the small number of nodes that can be analyzed, model checking provides valuable insight in the behavior of protocols for wireless sensor networks, insight that is complementary to what can be learned through the application of simulation and testing.

Acknowledgement We are most grateful to Frits van der Wateren for his patient explanations of the subtleties of the gMAC protocol. We thank Hernán Baró Graf for spotting a mistake in an earlier version of our model, and Mark Timmer for pointing us to the literature on communities in networks. Finally, we thank the anonymous reviewers for their comments.

References

[1] F.A. Assegei (2008): *Decentralized frame synchronization of a TDMA-based wireless sensor network.* Master's thesis, Eindhoven University of Technology, Department of Electrical Engineering.

[2] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi & M. Hendriks (2006): *Uppaal 4.0.* In: *Third International Conference on the Quantitative Evaluation of SysTems (QEST 2006),* 11-14 September 2006, Riverside, CA, USA. IEEE Computer Society, pp. 125–126.

[3] G. Behrmann, A. David & K.G. Larsen (2004): *A Tutorial on Uppaal.* In: M. Bernardo & F. Corradini, editors: *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures, Lecture Notes in Computer Science* 3185. Springer, pp. 200–236.

[4] J. Bengtsson, W.O.D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson & Wang Yi (1996): *Verification of an Audio Protocol with Bus Collision Using UPPAAL.* In: R. Alur & T.A. Henzinger, editors: *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, NJ, USA, *Lecture Notes in Computer Science* 1102. Springer-Verlag, pp. 244–256.

[5] R. Fan & N.A. Lynch (2006): *Gradient Clock Synchronization. Distributed Computing* 18(4), pp. 255–266.

[6] A. Fehnker, M. Fruth & A. McIver (2009): *Graphical Modelling for Simulation and Formal Analysis of Wireless Network Protocols.* In: M. Butler, C.B. Jones, A. Romanovsky & E. Troubitsyna, editors: *Methods, Models and Tools for Fault Tolerance, Lecture Notes in Computer Science* 5454. Springer, pp. 1–24.

[7] A. Fehnker, L. van Hoesel & A. Mader (2007): *Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks.* In: J. Davies & J. Gibbons, editors: *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings, Lecture Notes in Computer Science* 4591. Springer, pp. 253–272. Available at `http://dx.doi.org/10.1007/978-3-540-73210-5_14`.

[8] K. Havelund, A. Skou, K.G. Larsen & K. Lund (1997): *Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL.* In: *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA.* IEEE Computer Society, pp. 2–13.

[9] F. Heidarian, J. Schmaltz & F.W. Vaandrager (2009): *Analysis of a Clock Synchronization Protocol for Wireless Sensor Networks*. In: A. Cavalcanti & D. Dams, editors: *Proceedings 16th International Symposium of Formal Methods (FM2009), Eindhoven, the Netherlands, November 2-6, 2009, Lecture Notes in Computer Science* 5850. Springer, pp. 516–531.

[10] M.Z. Kwiatkowska, G. Norman & D. Parker (2004): *PRISM 2.0: A Tool for Probabilistic Model Checking*. In: *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST04)*. IEEE Computer Society, pp. 322–323.

[11] C. Lenzen, T. Locher & R. Wattenhofer (2008): *Clock Synchronization with Bounded Global and Local Skew*. In: *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*. IEEE Computer Society, pp. 509–518. Available at `http://dx.doi.org/10.1109/FOCS.2008.10`.

[12] L. Meier & L. Thiele (2005): *Gradient Clock Synchronization in Sensor Networks*. Technical Report 219, Computer Engineering and Networks Laboratory, ETH Zurich.

[13] M.E.J. Newman (2004): *Detecting community structure in networks*. *The European Physical Journal B* 38, pp. 321–330.

[14] R.M. Pussente & V.C. Barbosa (2009): *An algorithm for clock synchronization with the gradient property in sensor networks*. *Journal of Parallel and Distributed Computing* 69(3), pp. 261 – 265. Available at `http://www.sciencedirect.com/science/article/B6WKJ-4TYJTY5-1/2/38716d7d9f37c51edb8ba05e508fa2ce`.

[15] QUASIMODO (2009). *Preliminary description of case studies*. Available at `http://www.quasimodo.aau.dk/safe_html/internal/Deliverables/Final/Deliverable-5.2.pdf`. Deliverable 5.2 from the FP7 ICT STREP project 214755 (QUASIMODO).

[16] B. Sundararaman, U. Buy & A. D. Kshemkalyani (2005): *Clock synchronization for wireless sensor networks: a survey*. *Ad Hoc Networks* 3(3), pp. 281 – 323. Available at `http://www.sciencedirect.com/science/article/B7576-4FDMVS4-1/2/63ed40b032c6bf3ad5fca7fdcbe9e35a`.

[17] R. Tjoa, K.L. Chee, P.K. Sivaprasad, S.V. Rao & J.G Lim (2004): *Clock drift reduction for relative time slot TDMA-based sensor networks*. In: *Proceedings of the* 15th *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC2004)*. pp. 1042–1047.

[18] S. Tschirner, L. Xuedong & W. Yi (2008): *Model-based validation of QoS properties of biomedical sensor networks*. In: L. de Alfaro & J. Palsberg, editors: *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*. ACM, pp. 69–78.

[19] F.W. Vaandrager & A.L. de Groot (2006): *Analysis of a Biphase Mark Protocol with Uppaal and PVS*. *Formal Aspects of Computing Journal* 18(4), pp. 433–458. Available at `http://www.ita.cs.ru.nl/publications/papers/fvaan/BMP.html`.

[20] F. van der Wateren (2009). *Personal communication*.

[21] M. Woehrle, K. Lampka & L. Thiele (2009): *Exploiting Timed Automata for Conformance Testing of Power Measurements*. In: J. Ouaknine & F. W. Vaandrager, editors: *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Budapest, Hungary, September 14-16, 2009. Proceedings, Lecture Notes in Computer Science* 5813. Springer, pp. 275–290. Available at `http://dx.doi.org/10.1007/978-3-642-04368-0_21`.