Technical Report: Model Checking for Energy Efficient Scheduling in Wireless Sensor Networks[⋆]

Peter H. Schmitt and Frank Werner

Universität Karlsruhe (TU), Fakultät für Informatik
Institut für Theoretische Informatik
{pschmitt,werner}@ira.uka.de

Abstract. Networking and power management of wireless energy - conscious sensor networks is an important area of current research. We investigate a network of MicaZ sensor motes using the ZigBee protocol for communication, and provide a model using Timed Safety Automata. Our analysis focuses on estimating energy consumption by model checking in different scenarios using the Uppaal[11] tool. Special interest is devoted to the energy use in marginal situations that rarely occur and consequently might not be seen doing simulation.

1 Introduction

The technique of model checking has been successfully used in many application areas. It has proved particularly useful in very early design stages when only a model or a blueprint of the product is available. Using model checking tools flaws and errors have been revealed early, reducing costly changes in the later product design cycle.

In this paper we investigate the question whether the success story of model checking can be repeated in the area of low-energy sensor networks. We want to gain experience how these networks can be modelled. What kind of analysis should and can be performed? Common safety and liveness properties will certainly still play a role, although we rather focus on questions related to energy consumption. What is the minimal energy needed to reach a state with a given property? Can we formulate conditions that will guarantee that the live time of a sensor node is at least three month?

The plan of this paper is as follows. In the rest of this introduction we briefly introduce the theoretical framework of model checking. Section 2 describes our model for sensor networks, in Section 3 we present our results, and conclude with the usual wrap-up and suggestions for future research in Section 4.

[⋆] This work is funded by the BW-FIT Project ZeuS (Zuverlässige Informationsbereitstellung in energiebewussten ubiquitären Systemen).

What Model Checking is. Model checking is a formal method for automatically verifying system designs which has been applied to an impressive variety of areas. The method requires a model $\mathcal{M}$ of the system under investigation, a property φ that the system should have, and an algorithm to check whether $\mathcal{M}$ indeed satisfies φ. The method is very flexible since it is applicable to all systems that can be modelled as some kind of finite state machine, and is the most successful technology for formal verification.

Model checking has been applied to an impressive variety of application areas [10, 7, 13], and investigations into applications of model checking related to wireless sensor networks are just starting. Fortunately, in most applications there is a small set of properties one is interested in and only a small fraction of the temporal logic is needed.

The downside of the model checking method is that models have to be finite. A great deal of research went into techniques for reducing infinite state systems via guided abstractions into finite state systems or more precise, into finite state systems of moderate size. Otherwise that verification algorithm runs out of space giving no answer.

Model checking has been exercised in many applications. See the web pages of some of the more well known tools for details: SPIN[1], UPPAAL[2], PRISM[3], HyTech[4]. We refer the interested reader to the papers [10, 7, 13] to get a first impression of this work.

Uppaal The application scenario of wireless sensor networks where the majority of nodes is in sleep mode to fulfil the energy constraints, seems to suggest the use of probabilistic models, but at the beginning it was not clear how probabilistic features would enter into our formal analysis. We were thus looking for an easy-to-use, fairly efficient model checking tool, capable of using a cost function on transitions, having a notion of time.

Due to several other considerations we decided to choose for modelling the sensor network UPPAAL[11], an integrated tool environment for the design, simulation and verification of real-time systems. The tool is fairly efficient, and adequate for systems that can be modelled as a collection of non-deterministic processes communicating through shared variables, binary-, or broadcasting channels, having a finite control structure, and real-valued clocks.

In addition it is easy to use since the verification algorithm works fully automatically, and in case of a state found satisfying property φ, a trace reaching that very state is provided. For property φ to be checked, it has to be formalised in some propositional temporal logic. This may pose a challenge to people without the necessary skills and experience in formal logic.

[1] `http://spinroot.com/`

[2] `http://www.uppaal.com/`

[3] `http://www.cs.bham.ac.uk/~dxp/prism/`

[4] `http://embedded.eecs.berkeley.edu/research/hytech/`

Timed Automata – Syntax UPPAAL is based on the theory of Timed Safety Automata as presented in [8, 9]. The automata concept that is actually deployed in UPPAAL slightly differs, and extends this theoretical concept. First of all an UPPAAL automaton $\mathcal{A}$ consists of a set L of locations. In the automaton shown in Fig. 5b Down, Send, Idle, and Rcv are examples of locations. The initial location $\ell_0 \in L$ - in Fig. 5b this is the leftmost - is recognisable by the double circle. Next, automata may use local and global variables V ranging over finite subsets of integers or arrays of integers. Boolean variables are modelled as variables with range $\{0, 1\}$. In Fig. 5b e.g., the variables sid, $a[id]$, and q are employed. What is shown in this figure is, to be precise, a template for an automaton. Templates may be instantiated to obtain the automata that make up a system, as shown in Fig. 4a. Templates may contain constants that will be assigned concrete values when the template is instantiated. Fig. 5b contains e.g., the constant id that will be instantiated to an unique identity of a process. Further constants are N and MOD. Here N is the total number of nodes participating in the network. The reason why we use the constant MOD stems from the requirement imposed by all model checking approaches that the number of states should be finite. Any kind of counting – in our application we will count the number of collisions, number of packets send or received and also the energy consumption – has to be somehow truncated. The way we will do this is by limit counting to the numbers $0, \ldots MOD - 1$ and perform all arithmetic operations modulo MOD. As a reminder $m\%MOD$ is the unique number r, with $0 \leq r < MOD$ such that there is some k satisfying $m = k \times m + r$.

An important part in the specification of UPPAAL automata, that distinguishes them from simple finite state machines, is a set $\mathcal{C}$ of clocks. These are real-valued variables. In Fig. 5b only the clock variable t occurs. Clocks may be local or global and different clocks may show different times. Progress of time is the same however for all clocks. The next ingredient in the specification of an UPPAAL automaton $\mathcal{A}$ is a finite set Ch of channels comprising a pairwise synchronisation concept by a sending and a receiving part. In Fig. 5b e.g., the channels Syn and Col occur.

This brings us to the most prominent part of an UPPAAL automaton, its set E of edges. An edge may best be written as $\ell \xrightarrow{g,c,u} \ell'$, where $\ell, \ell' \in L$ are locations, g is a constraint on the clocks and integer variables called the *guard* of the edge, c is a synchronisation term of the form $C!$ or $C?$ with C a channel, and u is a set of variable updates or clock resets called the *update* of the edge. In Fig. 5b the left arrow from Idle to Rcv has guard $a[id] == N$, synchronisation term $Sync?$, and an update $(col := (col + 1)\%MOD,\ a[id] = -1)$. Guards, synchronisation terms and updates need not occur, in which case defaults are used.

To explain the semantics of an edge we first consider the case that no synchronisation term c is given, $\ell_1 \xrightarrow{g_1,u_1} \ell_1'$. The intuitive meaning then is: if an automaton $\mathcal{A}_1$ has active location ℓ_1, guard g_1 is satisfied then the update u_1 is performed, and $\mathcal{A}_1$ moves to location ℓ_1'. In case an action on channel c is given as $C!$ then for the edge to fire it is additionally required that there is another automaton $\mathcal{A}_2$ in the system with an edge $\ell_2 \xrightarrow{g_2, C?, u_2} \ell_2'$, such that $\mathcal{A}_2$ is in location ℓ_2 and g_2 is satisfied. Then $\mathcal{A}_1$ moves to ℓ_1', $\mathcal{A}_2$ moves to ℓ_2' and the updates u_1, u_2 are performed concurrently. The same explanation applies when $C?$ occurs in the edge of $\mathcal{A}_1$ and $C!$ at the edge in automaton $\mathcal{A}_2$. It is customary to call the automaton with the $C!$-edge the *sender* and its counterpart $C?$, the *receiver*. The above explanation shows that sender and receiver change their states synchronously. The use of binary synchronisation in UPPAAL is blocking, i.e. that the transitions can only be taken if sender and receiver do participate. In contrast to this, UPPAAL also offers the concept of broadcasting where one sender does synchronise with an arbitrary number of receivers, and the sending process is hereby never blocked.

So far nothing has been said about the progress of time. Edges in an automaton only offer possibilities of state changes. The out-going edge in Fig. 5a from the initial node carries the guard `t >=ActiveCycle` which means that the transition is enabled when the guard is true. It does not say that as soon as t gets greater than *ActiveCycle* the edge must fire.

This brings us to the last part of the specification of an UPPAAL automaton $\mathcal{A}$, the partial mapping I that attaches invariants to locations. Invariants are simple variable constraints on locations, that allow a automaton $\mathcal{A}$ to stay in this location as long as the invariant is not violated. Rephrasing the invariant-guard duo, it can be said that in UPPAAL a guard enables the transition to be taken and a invariant eventually forces an entered state to be left. In Fig. 5a the location Idle carries the invariant `t<=ActivePeriod`. If an instance automation of the template in Fig. 5a is in location Idle and `t=>ActivePeriod` then it must make a move out of Idle.

Furthermore locations can be marked with urgent - in which case time is not allowed to elapse or committed, putting an even severe constraint on states. Committed locations must be left immediately after entering and no other transition is permitted to be taken before. An automaton may stay in an urgent location ℓ_{urg} to perform another actions involving a committed or another urgent location. But all the actions before the one that leaves ℓ_{urg} will be considered to happen at the same instance of time.

So far we have not talked much about the operations that perform the updates, or that may occur in guard formulae. There are of course the usual arithmetic and Boolean functions. In addition the user of UPPAAL can define his own functions by providing programs that compute them, see Fig. 3 for examples. This completes the specification of UPPAAL automata $\mathcal{A} = (L, V, \mathcal{C}, Ch, E, I)$.

Timed Automata – Behaviour To explain the behaviour of an UPPAAL automaton or a system of UPPAAL automata, we need the concepts of *action* and *timed trace*. The automaton concept from [8, 9] comprises of a set Σ of actions. For UPPAAL automata this is a derived concept.

The behaviour of an UPPAAL automaton $\mathcal{A}$ is defined by the set $run(\mathcal{A})$ of all its possible *runs*. A run is a finite or infinite sequence $s_0, s_1, \ldots, s_i \ldots$ of *states*. A state in turn is a pair (ℓ, u) of a location $\ell \in L$, and a function u that associates values to all variables and clocks. Of course, we require that ℓ_0 is the initial state and u_0 assigns the initial values. If we look at a system of automata the state of the system is the pair $(\boldsymbol{\ell}, u)$ where $\boldsymbol{\ell}$ is a vector $(\ell^1, \ldots, \ell^k)$ of locations for all automata $\mathcal{A}_1, \ldots, \mathcal{A}_k$ in the system, and u assigns values to all local and global variables and clocks. A sequence $s_0, s_1, \ldots, s_i \ldots$ of states is in $run(\mathcal{A})$ if there is a timed trace $(t_1, b_1), \ldots, (t_i, b_i), \ldots$ that demonstrates that $\mathcal{A}$ can reach the given states in the given order. This is to say that for all i the automaton $\mathcal{A}$ can change from state $s_{i-1} = (\ell_{i-1}, u_{i-1})$ to $s_i = (\ell_i, u_i)$ via the timed action (t_i, b_i). This change comes in two parts. In the first part from (ℓ_{i-1}, u_{i-1}) to (ℓ_{i-1}, u'_{i-1}) only the clock variables change by $d_i = t_i - t_{i-1}$ (with $t_0 = 0$), i.e., u and u' differ only on variables $x \in \mathcal{C}$, and $u'_{i-1}(x) = u_{i-1}(x) + d_i$. The second part is the firing of the edge or the pair of edges b_i in state (ℓ_{i-1}, u'_{i-1}) with end states $s_i = (\ell_i, u_i)$ as already explained above.

Timed Automata – Property Checking One of the most popular query languages for the analysis of real-time systems is the Computation Tree Logic (CTL) [6, 2]. The most important properties are displayed in the following, where φ is a constraint on variables, clocks, or state labels:

$\mathbf{A}\Box\varphi$	invariantly φ	φ is satisfied by all states on all runs.
$\mathbf{E}\Diamond\varphi$	possibly φ	There is a state within a run that satisfies φ.
$\mathbf{A}\Diamond\varphi$	always eventually φ	In all runs there is a state satisfying φ.
$\mathbf{E}\Box\varphi$	potentially always φ	There is a run such that all its states satisfy φ.

The symbols $\mathbf{A}\Box$, $\mathbf{E}\Diamond$, $\mathbf{A}\Diamond$ and $\mathbf{E}\Box$ are called *temporal operators*. Some of the search trees[4] for the above mentioned properties are depicted in Fig. 1 for easier grasp. Note, that the UPPAAL query language does not allow nesting of temporal operators and consequently only a subset of CTL is permissible.

The property $\mathbf{E}\Diamond\varphi$ could also be interpreted as the existence of a reachable state, that satisfies φ. The reachability problem for Timed Safety Automata is – fortunately – decidable. This result depends crucially on the fact that guards and updates are of a simple form.

2 Model of Sensor Network

For being comparable to other studies we chose the MicaZ sensor motes manufactured by Crossbow as the models energy basis since they provide a versatile platform in particular for low-energy sensor networks [12]. The protocol for communication between the sensors is the ZigBee protocol[15] because it can be beneficially used, combining low transmission rates while fulfilling the criteria of being energy-conscious.

Out of the different topologies that exist we choose the *Mesh-Network* to be the most appropriate ones in our scenario. What we pursue is a Peer-to-Peer network consisting of only FFDs (full functional devices) since we aim on using a beacon disabled network. Device-to-device, and device-to-router communication is established using the CSMA/CA access on the common medium. So devices wake up in certain intervals send their recently gathered information, and fall back to sleep. The communication medium is represented as one channel on which all devices communicate, which is fixed over the analysis. This is intentionally done since it introduces collisions and related situations of interest. The more, it llows us to neglect the overhead arising from network maintenance like active-, passive scans, and channel changes because of high traffic with close-by networks. Apart from having bidirectional communication we restrict the model on passing information from the sensor devices to the network coordinator, and explicitly forbid a vice versa communication flow.

Uppaal Model. In our approach of building a sensor model using timed automata we aim at a homogeneous networking scenarios and will consider MicaZ motes in the role of Routers (ZR), and End Devices (ZED). The corresponding UPPAAL automata templates are shown in Fig. 5. Controllers are only capable of receiving information from the network. The Controller template (Fig. 5a) is thus obtained from the Router template, Fig. 5b, by leaving out the sending part of the automaton. The sensor motes, Fig. 5c, collect information being queued and transmit it to the routers.

As can be seen there are only subliminal differences between the three templates. The sensor nodes are the only devices which have the capability of col-

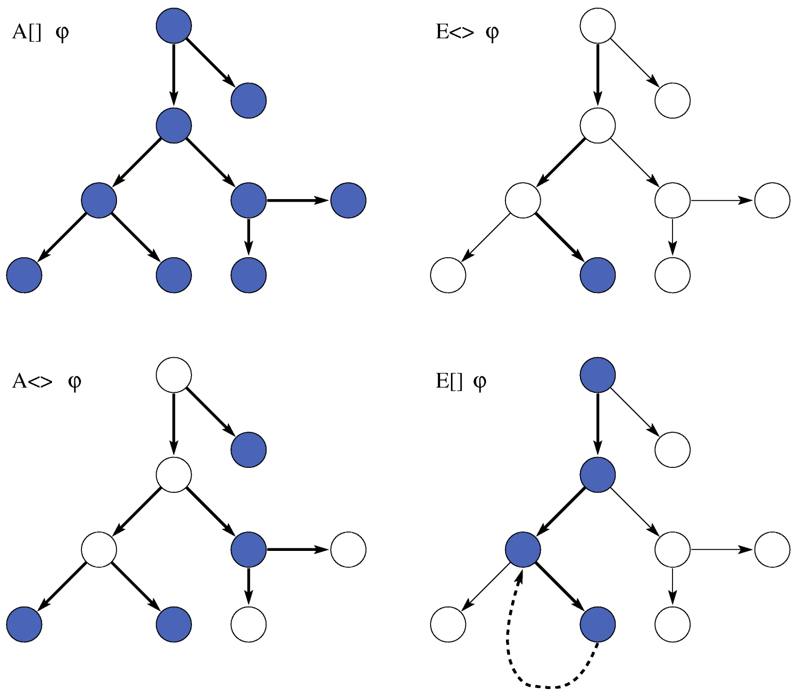


Fig. 1: Some possible temporal operators in the UPPAAL environment with satisfying states bluish highlighted.

lecting sensor values but in turn have no means to receive packets from the rest of the network. Routers can receive and forward packets, and finally the network controller has a mere capability of receiving packets. For the measure of energy consumed, a reference sensor, and a reference router are modelled. Due to this design issues the state space is kept small, retaining the essential functions each device is housing.

The power draw of the reference devices [1] is incorporated into the model, using values as shown in the table below (Tab. 1), and costs are accumulated whenever state changes occur.
Although the model makes no use of changing transmission rates during execution due to complexity considerations, the MicaZ transmission can be changed in advance accordingly to the distance matrix from Fig. 4b in concrete steps of $-10dBm$, $-5dBm$, and $0dBm$. To account for a very restricted state space, the models transmission rate is constant in the model but changed over different properties to imitate different scenarios. Higher transmission rates impose a higher energy consumption on each device, but simultaneously enable packets to reach the ZC using less hops along the network links. On the other side packets transmitted from other motes might collide more often in this case, and consequently the number of retransmissions before a packet's successful delivery to its destination is increased.

For obtaining a deeper understanding of our model, we consider the ZigBee reference controller from Fig. 5a and explain in particular the functions and essential variables used.

Each automaton is labelled with a unique ID. Whenever the globally modelled clock exceeds a value where a new round should be started (`t>=ActiveCycle`), the ZigBee controller is initiating a new cycle by waking up all devices from sleep-mode through broadcasting action GoIdle! over the network. Consequently all devices awake and capable ones update their energy consumption according to the state just left by use of the cost-function CE() cf. Fig. 2. Notice at this point that all synchronisation action within the model are broadcasting.

In addition, the sensor motes collect a sensor value which is queued at the sensor nodes (`q=(q+1)%MOD`). From now on each device is allowed to process and transmit data until the *Active* period is expired (`t<=ActivePeriod`). Note at this point the ZR and ZC have an active period prolonged by one time unit to account for network management and configuration messages. All devices have to leave state Idle whenever the state invariant labelling the idle state is violated. Sensor nodes can fall back to sleep earlier, that is if their sensor value is successfully transmitted to the network and before the `ActivePeriod` expired.

Next we will describe the action of sending in greater detail. The array `a[ ]` of length N denotes the availability of sensors and is manipulated by the transition updates during the operation of the network. For each i, $0 \leq i < N$ the array entry `a[i]` will be one of the values in $\{-1, 0, \ldots, N-1, N\}$. The intended meaning is such that `a[i] == j` if and only if a sending sensor j intends to send, and a node i within the neighbourhood of j is within a receivable range. The more we set `a[i]==-1` to signal that the channel within the neighbourhood of sensor i is idle, and `a[i]==N` to indicate that node i is experiencing a collision on the the channel caused by any of the neighbouring sensor notes.

A gadget whose channel is idle, i.e. a gadget satisfying `a[id]== -1`, may enter into state sending if $q > 0$, i.e. there is a queued message for sending. Simultaneously `t<=ActivePeriod-1`, stressing that there is still time to send and being heard by other devices. Besides from the update of the energy consumption CE(PIdle), function CheckAv(), as shown in Fig. 3d, is executed. This operation has no return value but changes the entries of array $a[]$. For all nodes i within reach, the function call identifies potential receivers by setting `a[i]==id` if i is idle and sets `a[i]==N` to signal a collision before moving to location send. In case of a collision, it is possible that either a synchronisation of another group of automata just took place in which case all receivers send a Col signal or that there are some receivers waiting for another gadget to send data.

```
void CE(int e){
    //Compute Energy of Mote modulo CMOD
    c = (c+e) % CMOD;
}
```

Fig. 2: Definition of the UPPAAL cost function.

If no collision is detected in state Send (`!Checkid(id)`), the automaton in commencing with the synchronisation after declaring itself to be sending (`sid=id`). By the use of the global variable `sid` an sending process is indicating its intention to send.

Table 1: Energy consumed by the MicaZ Sensor in each state.

State	Proc Draw[μA]	TX/RX Draw[μA]	Remarks
PDown	15	1	Energy draw in sleep
PSleep	8 000	1	Proc. up, Tx/Rx down
PIdle	8 000	20	Proc. up, Tx/Rx up
PSnd1	8 000	11 000	sending at $-10dBm$
PSnd2	8 000	14 000	sending at $-5dBm$
PSnd3	8 000	17 400	sending at $0dBm$
PRcv	8 000	19 700	receiving mode

If afterwards any automaton received the data (`receivers[id]>0)`), the packet is removed from the sender's queue (`q=(MOD+q-1)%MOD`) and the channel is cleared by function CleanAll(id) in figure 3c. Otherwise the packet is kept in the queue. This modelling is legitimate since only one process can send at a time since the whole sending procedure takes places within the two committed states in which no other transmission can take place.

For the receiving side, a device is transitioning from state Idle to Receive if it is either able to synchronise with action Sync? and no collision is detected (`sid==a[id]`) in which case the packet is acknowledged at the sending side (`received[sid]=(received[sid]+1)%MOD`), or a packet collision occurred and the sender is informed (`a[id]=-1`).

ZigBee Protocol. For being comparable to results gained through practical experiments the model is designed as proposed in the ZigBee specification [15] for homogeneous networks underlying a tree topology. The setup is as shown in Fig. 4a. Digits in parenthesis indicate the process number used for identification later on in the verification part. Whenever a sensor end device (ZED) is waking up, it is forwarding the recently collected sensor value to the router (ZR) and the going back to sleep again. The routers in turn pass the packet along the network link - possibly using other routers - to the ZigBee network controller (ZC) which is the root of each ZigBee network and unique. Routers are the only devices in our model that embody a sending and receiving side.

Since most energy is preserved in sleep mode - where processor, and on-board transmission unit are shut down - we target an average duty cycle of 1%[12] by defining *ActivePeriod* as 1 and *ActiveCycle* as 100. Explicitly note at this point that we model a beacon disabled network without a contention free period, i.e. that collisions can always occur. Distances between respective entities are modelled using the distance matrix from Fig. 4b to determine communication flow within the sensor network as well as the number of hops a packet as to undergo until arriving at its destination.

By modelling a CSMA/CA like feature, packet collisions are avoided by devices within the same transmission range where possible. Although being costly, a device is sensing the channel for traffic before sending data and as such collisions can only occur by participants outside the transmission range which don't detect ongoing transmissions.

Cost Estimation. Since special interest is devoted to the estimation of costs our automata should offer means to keep track of e.g. energy consumption. To avoid unnecessary blow-up of the state space not every mote automaton is equipped with the *cost estimation* function `CE`, Fig. 2. Rather, we singly use one reference sensor node [cf. Fig. 5c], and one reference router [cf. Fig. 5b] in our system of automata. Only these are equipped with `CE` function. We chose not to estimate cost for the controller mote, since typically this may have unlimited energy supply. The function `CE` adds costs from Tab. 1 for state changes by the automata,

```
bool CheckID(id_t id){
    bool rv=false;
    int i=0;
    for (0; i<N; i++)
        if ((dist[id][i]<=TXpowerLow) & (a[i]==N))
            rv=true;
    return rv;
}
```

(a) Function `CheckID()` checking whether any device within range of device `id` (`dist[id][i]`) is able to receive in which case `true` is returned.

```
void Clean(id_t id){
    int i=0;
    for (0; i<N; i++)
        if ((dist[id][i]<=TXpowerLow) & (a[i]==id))
            a[i]=-1;
}
```

(b) `Clean(id)` is clearing the channel for devices that carry the flag of sensor `id`.

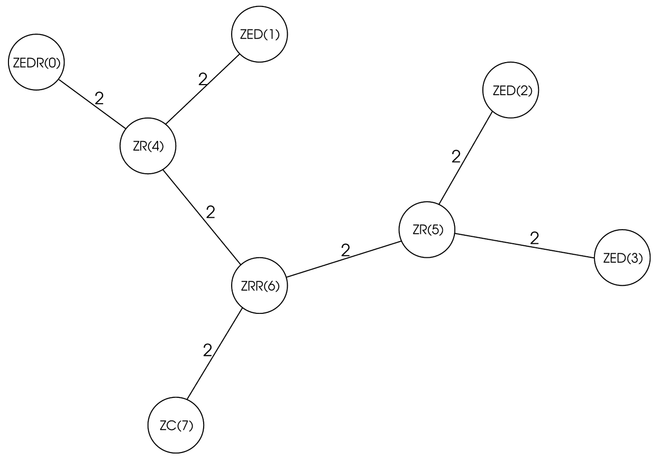
```
void CleanAll(id_t id){
    int i=0;
    for (0; i<N; i++)
        if (dist[id][i]<=TXpowerLow)
            a[i]=-1;
}
```

(c) Function `CleanAll(id)` clears the channel of all gadgets within the transmission range.

```
void CheckAv(int id){
    int i;
    for (0; i<N; i++)
        if (dist[id][i]<=TXpowerLow)
            //Put Senders ID on Receivers Channel
            if (a[i]==-1) a[i]=id;
        else
            //Put Collision on Receivers Channel
            if(a[i]>-1) a[i]=N;
}
```

(d) Function `CheckAv(id)` does check the availability of each gadget within the transmission range by setting a flag on array `a[]`.

Fig. 3: Definition of the UPPAAL functions.



$$\begin{pmatrix} 0&4&8&8&2&6&4&6 \\ 4&0&8&8&2&6&4&6 \\ 8&8&0&4&6&2&4&6 \\ 8&8&4&0&6&2&4&6 \\ 2&2&6&6&0&4&2&4 \\ 6&6&2&2&4&0&2&4 \\ 4&4&4&4&2&2&0&2 \\ 6&6&6&6&4&4&2&0 \end{pmatrix}$$

(a) Scenario with reference sensor (ZEDR), sensors (ZED), routers (ZR), reference router (ZRR) with cost function, and ZigBee Controller (ZC). Edges are labelled with distances and numbers in parenthesis denoted process.

(b) Distance matrix for the selected scenario.

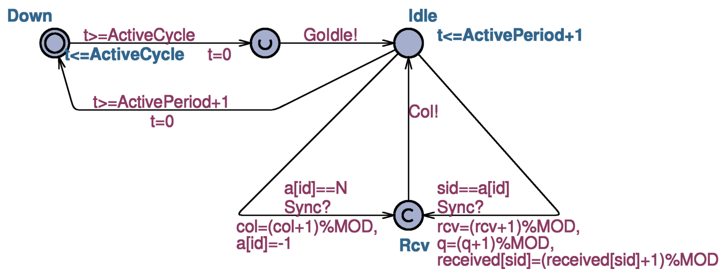
Fig. 4: Scenario settings underlying the analysed sensor network.

giving some observables at hand. Respectively adopted to this design is the cost of leaving state Down. Using the cost $PDown$ multiplied by 99 time units gives an accurate energy consumption for the time spent in this state, totalling the transition cost to $PDown' = 1\,548 \mu A$. This approach is used since clock values cannot be incorporated in arithmetic calculations with numbers of type integer in the Uppaal version at hand.

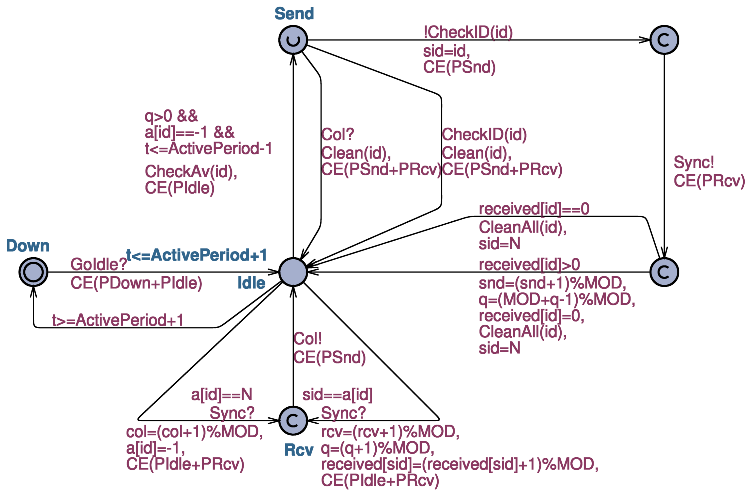
For finding proper estimates to account for the energy used, we would require Uppaal to use the costs from table 1 multiplied by the time spend in the corresponding state which is currently not feasible due to type constraints.
Besides this we also investigated the used of priced timed automata [5, 14] as proposed in many studies[3]. The use of the Uppaal Cora [5] which is using the technique of cost optimal reachability analysis in linearly priced timed automata did not deliver the promising results and completely failed in the example at hand. Although several case studies did provide fruitful results, it failed to do so here when working with increased complexity as in our example.

Means of Reducing the State Space By introducing several efficiency means, the state space could be condensed far enough to verify essential properties. As such variables are declared as *meta* that will not be considered when building the state vector of Uppaal, and thus the state space is kept small. Another technique frequently used to avoid unbounded variables is by defining a modulo-class

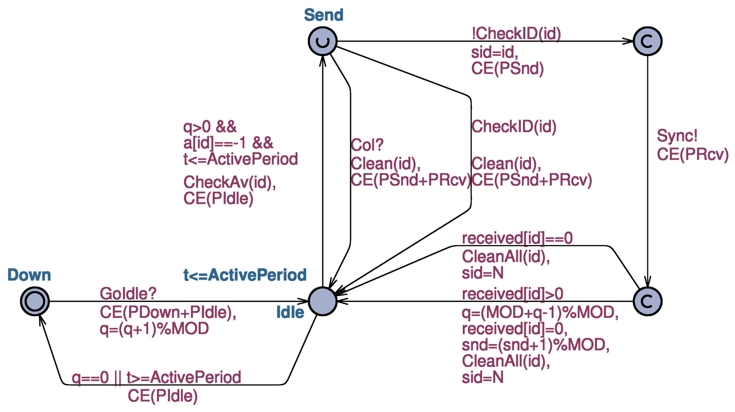
[5] `http://www.cs.aau.dk/~behrmann/cora/`



(a) Template for network controller



(b) Template for reference routers



(c) Template for reference sensor mote

Fig. 5: Uppaal timed automata model of the sensor network.

of variables by a constant MOD. Consequently an upper bound is introduced avoiding a state-explosion by unbounded values. Besides this, variables are left out where not necessarily used, depending on the property of interest.

3 Verification Results

Properties are verified using a hash table size of 512MB for state hashing in the Uppaal tool setting, and giving the shortest path for some satisfying property, needed for finding the lowest cost. Before the outcomes of different properties are tested using the new model, it is tested for being free of deadlocks ($A\Box\neg deadlock$) to assure plausible sound modelling and sanity. For all experiments the state space is bound by fixing variable MOD to appropriate values. The transmission rate is increased over different scenarios from $-10dBm$, $-5dBm$, to $0dBm$, augmenting the theoretical coverage.

3.1 Energy Considerations

Starting with a deadlock free model, energy considerations are obtained by searching the state space spanned by the model for properties as specified by the user. Whenever a satisfying state is found, a path is generated that shows the transitions taken until the state is reached. All experiments conducted here investigate the use of energy of the reference sensor and router under different scenarios. The desired properties are checked by definition of CTL[6, 2] formulae.

Sensor Devices. The first experiment conducted is observing the power drawn by the reference MicaZ sensor under the following property:

$$E\Diamond\, ZC(7).rcv = 1 \& ZEDR(0).snd = 1$$

Or state more verbally :"How much energy does the reference node spend by sending a packet - expressed in a formal property as ($ZEDR(0).snd = 1$) - which is routed through the network and finally received by the controller ($ZC(7).rcv = 1$)?" By use of the temporal operator $E\Diamond\ \varphi$ ("Does there exist a path such that φ does eventually hold in the future"), the shortest path is returned as defined by the appropriate strategy. Tab. 2 captures the energy drawn for the above property while varying the transmission reach.

Table 2: Energy consumed by the sensor device for different ranges of coverage.

dist.	TX[dBm]	Property	power use [μA]
2	-10	$E\Diamond\, ZC(7).rcv = 1 \& ZEDR(0).snd = 1$	64 300
4	-5	$E\Diamond\, ZC(7).rcv = 1 \& ZEDR(0).snd = 1$	67 300
6	0	$E\Diamond\, ZC(7).rcv = 1 \& ZEDR(0).snd = 1$	70 700

ZigBeeRouters. After having studied the energy use by sensor gadgets, a further step is to investigate the costs that occur at the routing devices, since they need more power due to higher activity. For this scenario the reference router ZRR from Fig. 5b has been chosen, since it interlinks the controller with the network, and is hence most critical to energy constraints. The analysis observes the energy consumption by the router using different transmission rates, and increasing collisions occurring at the router over the experiments. Results are illustrated in table 3 below.

As expected, by increasing the transmission rates more devices in the network are capable of over-leaping the reference router node, thus preserving the ZRR's energy, and enable a faster delivery of packets to the ZC at the root.

Table 3: Energy consumed by the reference ZigBee router under different scenarios.

Property	power use $[mA]$		
Tx/Rx in dBm with (distances)	-10 (2)	-5 (4)	0 (6)
$E\Diamond ZC(7).rcv = 1$	120	126	46
$E\Diamond ZC(7).rcv = 1 \& ZRR(6).col = 1$	156	104	107
$E\Diamond ZC(7).rcv = 1 \& ZRR(6).col = 2$	211	104	168
$E\Diamond ZC(7).rcv = 1 \& ZRR(6).col = 3$	266	162	229

4 Conclusion

Our experiments showed that the timed automata model presented in Section 2 is a good basis for the analysis of energy consumption of sensor motes within an arbitrary scenario. Special emphasis is hereby on the investigation of marginal or borderline situations which rarely occur in simulation and can thus be exhaustively analysed using the here presented approach.

Furthermore we were able to determine cost optimal timings for specific schedules. The features provided by the Uppaal tool proved to be flexible enough to formulate queries involving the estimation of energy consumption. Especially useful was the fact that Uppaal offers a built-in concept of multi-cast. Although the verification is restricted due to limitations of Uppaal, we believe that the model can be adopted to suit an even deeper analysis than shown here. Especially by accompanying the verification techniques pursued here with more realistic data, a deeper understanding of routing, contentions, and the energy related can be obtained.

So far we have not made a serious attempt to use the counter measures recommended in the Uppaal tutorial[4] to curb state explosion. It will also be promising to explore the potentials of other tools, and to look into infinite model checkers.

References

1. Micaz data sheet - wireless sensor networks. `www.xbow.com`.
2. Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. 1990.
3. Gerd Behrmann, Ed Brinksma, Martijn Hendriks, and Angelika Mader. Production scheduling by reachability analysis - a case studymodel-checking for real-time systems. Ametist Project.
4. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. Technical report, Department of Computer Science, Aalborg University, Denmark, November 2004.
5. Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. Technical report, BRICS, Aalborg University, Denmark.
6. E. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In J.W. deBakker, W.P. deRoever, and G. Rozenberg, editors, *Proc. Workshop on Linear Time, Branching Time, and Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 257–268. Springer, 1988.
7. Sinem Coleri, Mustafa Ergen, and T. John Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 98–104, New York, NY, USA, 2002. ACM Press.
8. Thomas A. Henzinger, Xavier Cicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proc. 7th Annual IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.
9. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Journal of Information and Computation*, 111(2):193–244, 1994.
10. YoungMin Kwon and Gul Agha. Performance evaluation of sensor networks: A statistical modeling and probabilistic model checking approach. In *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 2006.
11. Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1997.
12. Ciaran Lynch and Fergus O'Reilly. Processor choice for wireless sensor networks. In *Proc. 1st Workshop on Real-World Wireless Sensor Networks REALWSN*, number T2005:09 in SICS Technical Reports, pages 58–62. SICS, Stockholm, Sweden, 2005.
13. G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Aspects of Computing*, 17(2):160–176, August 2005.
14. J.I. Rasmussen, Kim G. Larsen, and K. Subramani. *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, chapter Resource-Optimal Scheduling Using Priced Timed Automata, pages 220–235. Springer, 2004.
15. ZigBee specification. Zig-Bee Document 053474r06, Version 1.0, June 2005.