	ICT-FP7-STREP-214755 / QUASIMODO January 26, 2009 **Page 1 of 24**	

Project no.: **ICT-FP7-STREP-214755**

Project full title: **Quantitative System Properties in Model-Driven Design**

Project Acronym: **QUASIMODO**

Deliverable no.: D5.5

Title of Deliverable: Case Studies: Models

Contractual Date of Delivery to the CEC:	**Month 12**
Actual Date of Delivery to the CEC:	**Month 12 (01. February 2009)**
Organisation name of lead contractor for this deliverable:	**P04 RWTH Aachen**
Author(s):	**Henrik Bohnenkamp, Joost-Pieter Katoen, Kai Mittermüller, Holger Hermanns, Julien Schmaltz, Faranak Heydarian, Frank Cassez, Haidi Yue**
Participants(s):	**P01, P02, P03, P04, P05, P06, P08, P10**
Work package contributing to the deliverable:	**WP5**
Nature:	**R**
Version:	**1.0**
Total number of pages:	**24**
Start date of project:	1 Jan. 2008 **Duration:** 36 month

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

This deliverable describes the models used in the QUASIMODO project to tackle the case studies introduced in Deliverable 5.2: *Preliminary Description of case studies*.

Keyword list: Case Studies, Models.

Contents

Abbreviations 3

1 Introduction 4

2 Hydac: Accumulator Charge Controller 5
2.1 SU 5
2.1.1 Participants 5
2.1.2 Model 5
2.1.3 First Results 8
2.2 CNRS, AAU, CFV 10
2.2.1 Participants 10
2.2.2 Models 10
2.2.3 First Results 14

3 CHESS: Wireless Sensing 15
3.1 ESI/RU 15
3.1.1 Participants 15
3.1.2 Model 15
3.1.3 First results 17
3.2 RWTH Aachen 19
3.2.1 Participants 19
3.2.2 Model 19
3.2.3 First Results 21

Bibliography 24

Abbreviations

AAU: Aalborg University, DK (P01)

ESI: Embedded Systems Institute, NL (P02)

ESI/RU: Radboud University Nijmegen, NL

CNRS: National Center for Scientific Research, FR (P03)

RWTH: RWTH Aachen University, D (P04)

SU: Saarland University, D (P05)

CFV: Centre Fèdèrè en Vèrification, B (P06)

CHESS: CHESS (P08)

HYDAC: Hydac (P10)

1 Introduction

This deliverable provides first insight into how the Quasimodo partners tackle the case studies introduced in Deliverable 5.2 [7]. In particular, it is described how the systems are modelled and to what purpose. First results, as far as they exist, are presented.

Two case studies of the proposed four are currently under scrutiny by the project partners: the gMAC protocol, proposed by CHESS, and the *Accumulator Charge Controller* case study, proposed by Hydac.

2 Hydac: Accumulator Charge Controller

For a description of this case-study we refer to [7].

2.1 SU

2.1.1 Participants

- Holger Hermanns, Saarland University.
- Kai Mittermüller, Hydac/Saarland University.

2.1.2 Model

The Hydac Case (Accumulator Charge Controller) has been modelled with the Simulink and Stateflow formalisms from the MathWorks.

Matlab-Simulink model. This case study is based on a product which has been developed by HYDAC, but is not yet available on the market. The problems and tasks described here for this concrete product are also easy transferable to other products, so the HYDAC has a great interest in the knowledge transfer provided by this EU-project. The product is an accumulator-charge controller (ACC) which optimises the energy and the wear of the used components, especially the pump. The Matlab-model was built to compare our newly developed controller with the existing one. The overall system is depicted in Figure 1.

The consumers are modelled as a "Repeating Sequence Block". This Block creates a recurring curve which represents the consumption of oil (Figure 2). The x-axis is the time in seconds, and the y-axis the consumption in l/s. The recurrence allows us to simulate successive repeating cycles. The "Accumulator-block" is again a Simulink model, which is depicted in Figure 3.

This model describes the physical behaviour in the accumulator. First we calculate the volume change (for the gas) from the state of the pump and the current consumption. This change is the derivation of the volume, hence the next block integrates this change. Thereby we assume an initial value of the gas volume of 40 litre. To get the oil volume we simple subtract the gas volume from the total volume of the accumulator. Based on the pressure of the oil we can calculate the pressure of the gas and from this pressure the energy consumption.

Now we take a look at the other controllers in our model: The "2-point (sl)" controller in Figure 4(a) contains a 2-point-controller as Simulink model. The "2-point (c)" controller in Figure 4(a) also describes a 2-point-controller, but as a Stateflow model which works with discrete time (clocked by "clock-block") and which uses C-functions. The clock is necessary to call the C-functions and is more realistic then a continuous time controller. The clock-rate is 10ms, which is a normal sample rate in hydraulics.

The third controller "ACC (c)" in Figure 5 contains the new accumulator charge control. For this controller, an additional signal "cycle" is necessary, which states that the next cycle starts. This signal is also available at real machines.

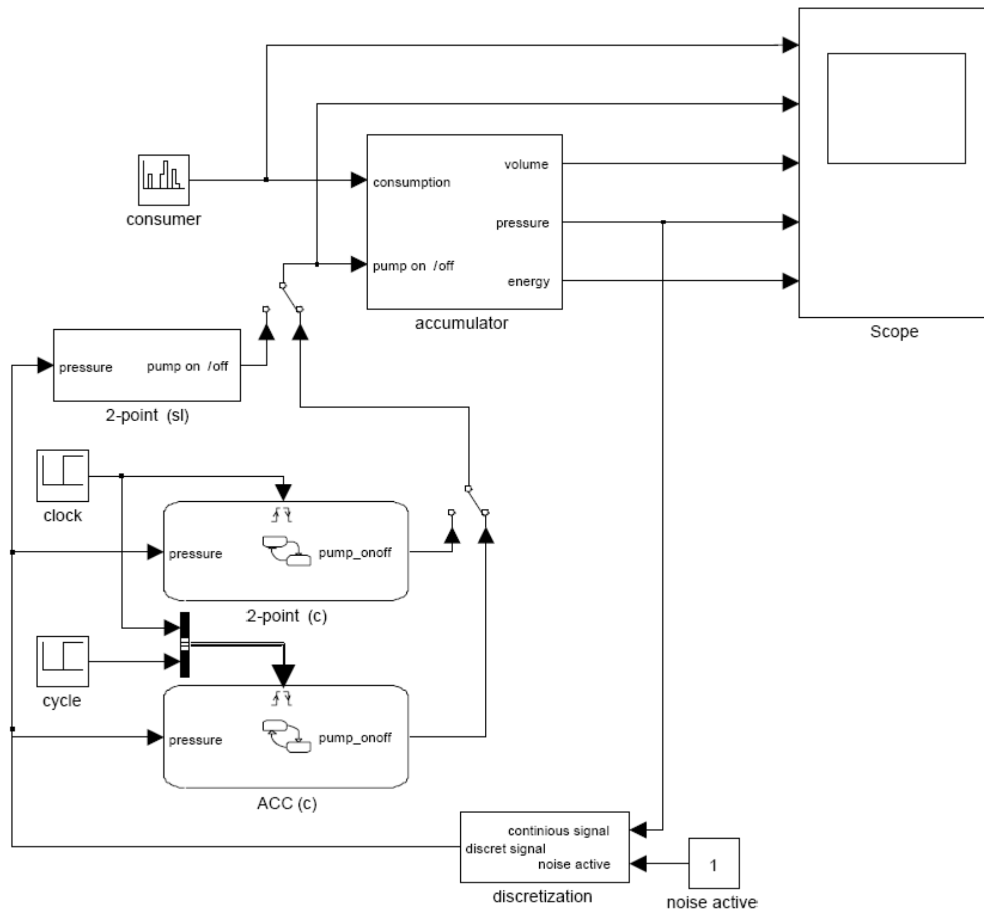


Figure 1: System

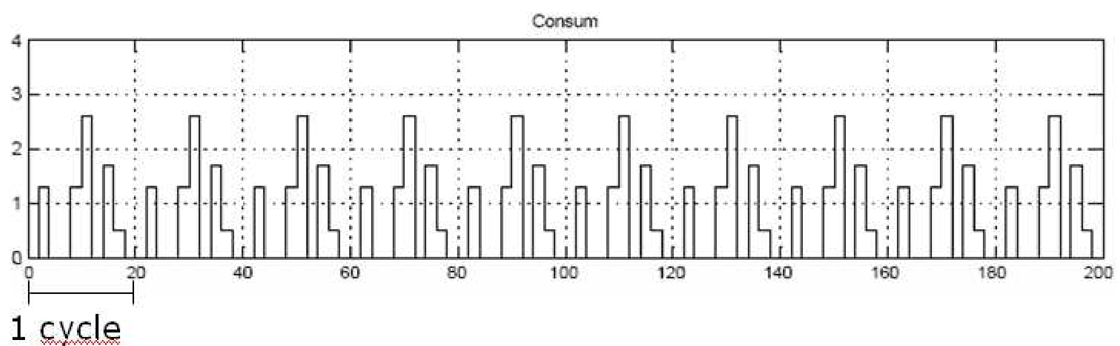
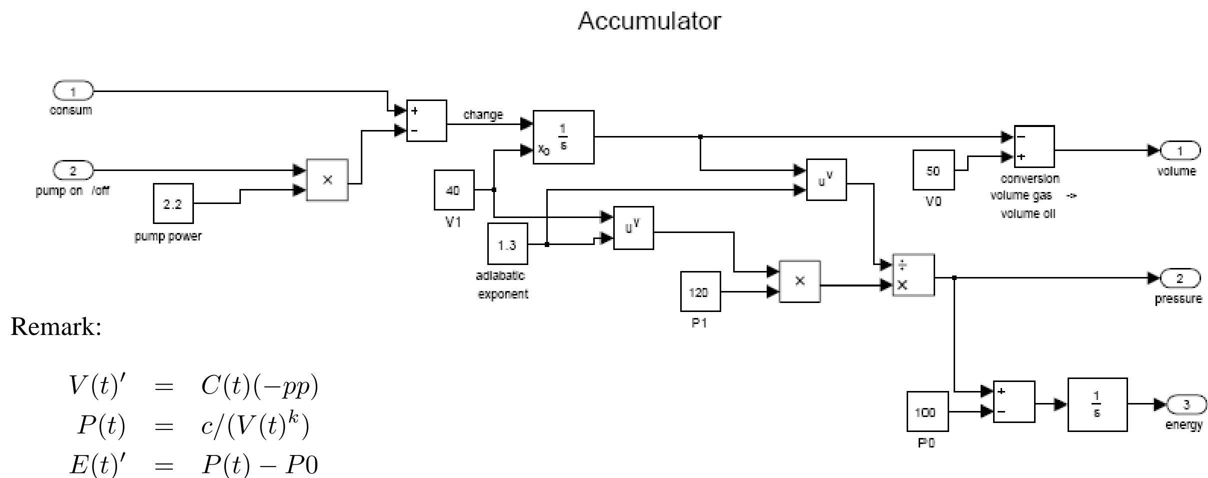


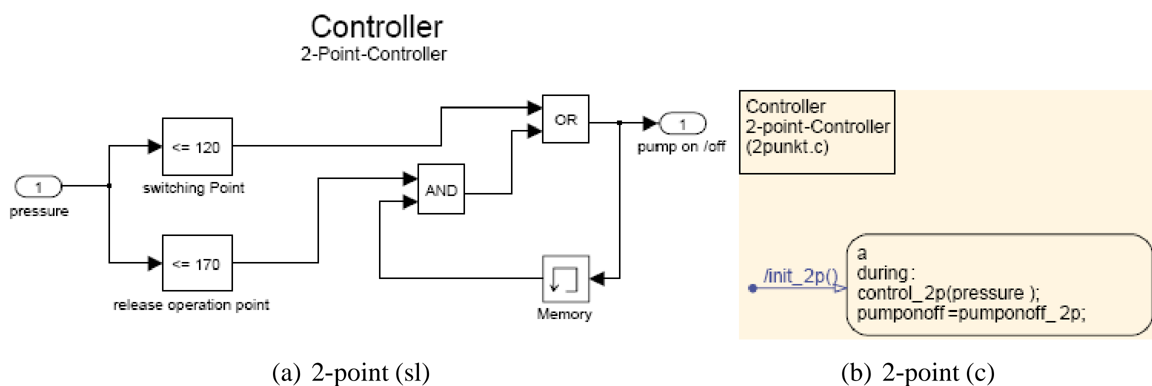
Figure 2: Consumer



Remark:

$$
\begin{aligned}
V(t)' &= C(t)(-pp) \\
P(t) &= c/(V(t)^k) \\
E(t)' &= P(t) - P0
\end{aligned}
$$

Figure 3: Accumulator



(a) 2-point (sl) (b) 2-point (c)

Figure 4: 2-point controllers

On the basis of this automaton the single phases of the accumulator charge controller can be recognised. The first state is the “cycle recognition”. In this state the ACC uses a 2-point controller to steer the pump and measures the pressure and the pump state. From these values it is possible to calculate the volume curve without the pump. Between these two cycles the function “optimize_acc()” calculates the optimal switching points for the pump, which are the basis for steering the next cycle. This happens with the function “control_acc()”. Equal to the cycle recognition the pressure and the pump state get recorded, too. The third state is a backup system, for the case that the pressure leaves the safe area. This is necessary, if an unusual event occurs (e.g. leakage). The backup system uses a 2-point controller.

To make the system more realistic, the model contains additionally a “Discretization-block” (Figure 6). This block discretises the pressure and allows on the other side to add noise. The discretisation of the pressure is done with 12 bit, which is the default measuring accuracy in hydraulics. In our case the range from 0 to 400 bar is split in 4096 possible values. The noise is created by a “Band-Limited White Noise”-block and can be turned on and off independently of

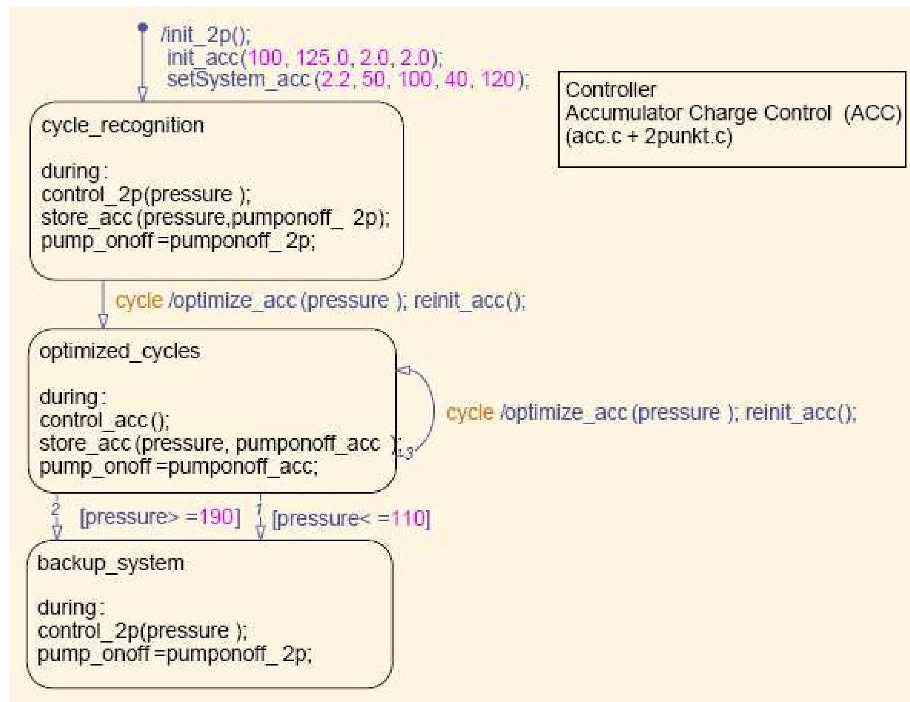


Figure 5: ACC

the discretisation in the system (Figure 1).

2.1.3 First Results

The Simulink-Stateflow models enabled us to get various insights into the functioning of the different systems. For that, we used the simulation capabilities provided by Simulink, on a variety of model instance with different consumer profiles, pumps and accumulator settings. In all the simulation studies carried out, the following observations were made:.

1. The 2-point controller always keeps the pressure in safe margins.
2. The ACC controller always keeps the pressure in safe margins.
3. The ACC controller always uses considerable less energy than the 2-point controller.

Thus we were able to experimentally validate—but not formally show—some of our most basic system requirements. The third experimental observation is exemplified in the chart in Figure 7.

This work and further details are reported in [9].

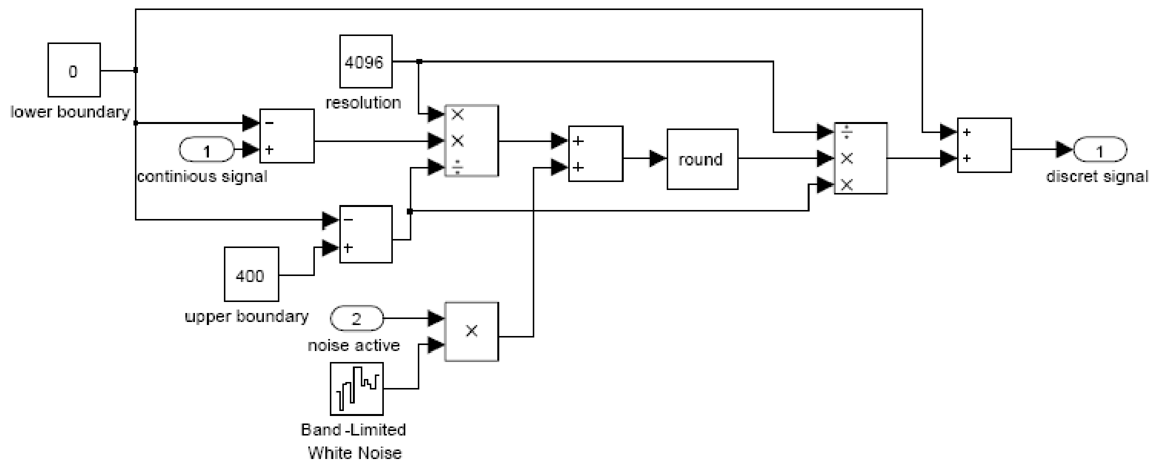


Figure 6: Discretisation

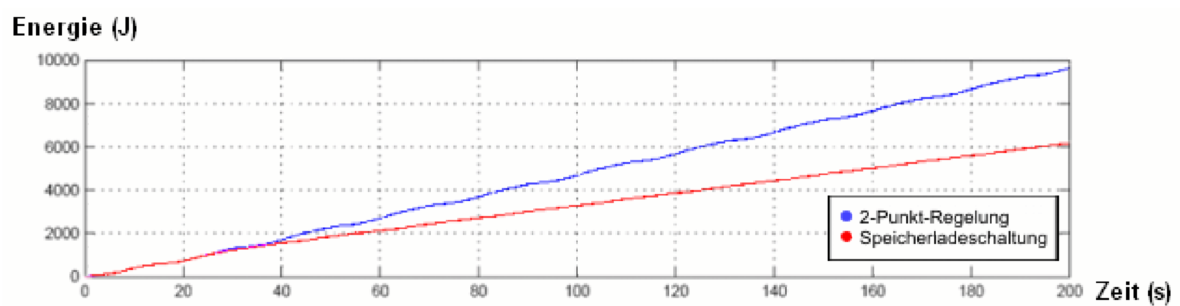


Figure 7: Energy consumption 2-point controller/ACC

2.2 CNRS, AAU, CFV

2.2.1 Participants

- Franck Cassez, National ICT Australia & CNRS, Sydney, Australia
- Jan J. Jessen and Kim G. Larsen, CISS, CS, Aalborg University, Denmark
- Jean-François Raskin, CS Department, Université Libre de Bruxelles, Belgium
- Pierre-Alain Reynier, LIF, University of Marseille & CNRS, UMR 6166, France
- Kai Mittermüller, Hydac.

2.2.2 Models

To solve the Hydac control problem, we use three complementary tools for three different purposes:

- For the synthesis phase, we construct a (game) model of the case study and compute a controller with UPPAAL-TIGA [1];
- To verify the robustness of our controller we use PHAVER [6] and embed our synthesised controller into a continuous environment modelled by a hybrid automaton.
- Finally, performance-wise, we use SIMULINK [10] to compare the synthesised controller with the ones provided by Hydac: the 2-point and the Smart controllers (the latter being described in Section 2.1).

The oil consumption of the machine is cyclic. The cycle of consumptions, as given by the Hydac company, is depicted in Fig. 8.

Detailed Description of the System. Each period of consumption is characterised by a rate of consumption (expressed as a number of litres per second), a date of beginning and a duration. We assume that the cycle is known *a priori*: we do not consider the problem of identifying the cycle (which can be performed as a pre-processing step). The control strategy must allow the machine to operate for an arbitrarily large number of cycles. At time 2, the rate of the machine goes to $1.2l/s$ for two seconds. From 8 to 10 it is 1.2 again and from 10 to 12 it goes up to 2.5 (which is more than the maximal output of the pump). From 14 to 16 it is 1.7 and from 16 to 18 it is 0.5.

Even if the consumption is cyclic and known in advance, the rate is subject to *noise*: if the mean consumption for a period is $c\ l/s$, in reality it always lies within that period in the interval $[c-\epsilon, c+\epsilon]$, where ϵ is fixed to $0.1\ l/s$. This property is noted F.

The volume of oil within the accumulator is initially equal to $10\ l$. The pump is either *on* or *off*, and we assume it is initially *off*. The operation of the pump must respect the following *latency* constraint: there must always be two seconds between any change of state of the pump, i.e. if it

is turned *on* (respectively *off*) at time t, it must stay on (respectively *off*) at least until time $t + 2$, we note P_1 this property. When it is *on*, its *output* is equal to $2.2l/s$. Note that as the power of the pump is not always larger than the demand of the machine during one period of consumption (see Fig. 8 between 10 and 12), some extra amount of oil must be present in the accumulator before that period of consumption to ensure that the minimal amount of oil constraint (requirement R_1) is not violated[1].

The controller must operate the pump (switch it on and off) to ensure the following two main requirements:

- (R_1): the level of oil $v(t)$ at time t (measured in litres) into the accumulator must always stay within two *safety* bounds $[V_{min}; V_{max}]$, in the sequel $V_{min} = 4.9l$ and $V_{max} = 25.1l$;
- (R_2): a large amount of oil in the accumulator implies a high pressure of gas in the accumulator. This requires more energy from the pump to fill in the accumulator and also speeds up the wear of the machine. This is why the level of oil should be kept minimal during operation, in the sense that $\int_{t=0}^{t=T} v(t)$ is minimal for a given operation period T.

While requirement (R_1) is a *safety requirement* and so must never be violated by any controller, (R_2) is an *optimality* requirement and will be used to compare different controllers.

To summarise, the controller to design must turn on and off the pump at the appropriate points in time while respecting the *latency* of the pump (property P_1) to ensure that requirements R_1 is satisfied, even under the fluctuations F within the cyclic consumption phase, for an arbitrarily long period of time. Moreover we should try to minimise the accumulated oil during each cycle (requirement R_2). Because the consumptions are subject to noise, it is necessary to allow the controller to check periodically the level of oil in the accumulator (as it is not predictable in the long run). Nevertheless, the controller should exploit the cyclic nature of the consumption to optimise the level of oil. So, we will also allow our controllers to take control decisions at predefined instant in time during the cycle using timers.

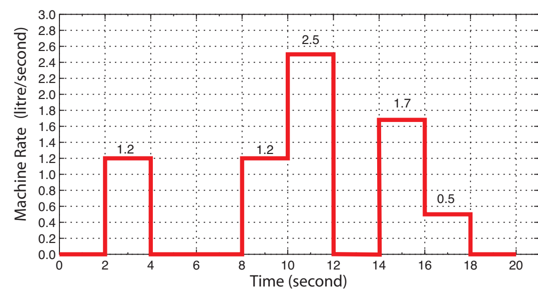


Figure 8: Cycle of the Machine.

The UPPAAL-TIGA Model. To keep the model simple enough, we have designed a model which: (a) considers one cycle of consumption; (b) uses an abstract model of the fluctuations of the rate; (c) uses a discretisation of the dynamics within the system. Second, to make sure that the winning strategies that will be computed by UPPAAL-TIGA are implementable, the states of our game model only contain the following information, which can be made available to an implementation:

- the volume of oil at the beginning of the cycle;
- the ideal volume as predicted by the consumption period in the cycle;

[1] It might be too late to switch the pump on when the volume reaches V_{min}.

- the current time within the cycle;
- the state of the pump (*on* or *off*).

First, we discretise the time w.r.t. ratio stored in variable D, such that D time units represent one second. Second, we represent the current volume of oil by the variable V. We consider a precision of $0.1l$ and thus multiply the value of the volume by 10 to use integers. This volume evolves according to a rate stored in variable V_rate and the accumulated volume is stored in the variable V_acc. Finally, we also use an integer variable time which measures the global time since the beginning of the cycle.

The model for the behaviour of the machine is represented on Fig. 9(a). Note that all the transitions are uncontrollable (represented by dashed arrows).When a time at which the rate of consumption changes is reached, we simply update the value of the variable V_rate. The additional central node called bad is used to model the uncertainty on the value of V due to the fluctuations of the consumption. The function Noise checks whether the value of V, if modified by these fluctuations, may be outside the interval $[V_{min} + 0.1, V_{max} - 0.1]$ [2]. The function final_Noise checks the same but for the volume obtained at the end of cycle and against the interval represented by V1F and V2F.

The model for the pump is represented on Fig. 9(b). The transitions are all controllable (plain arrows). The pump simply consists of two locations representing whether the pump is ON or OFF. Moreover, the latency constraint[3] P_1 is expressed using the clock z. An additional integer variable i is used to count how many times the pump has been started on. We use parameter N to bound this number of activations, which is set to 2 in the following.

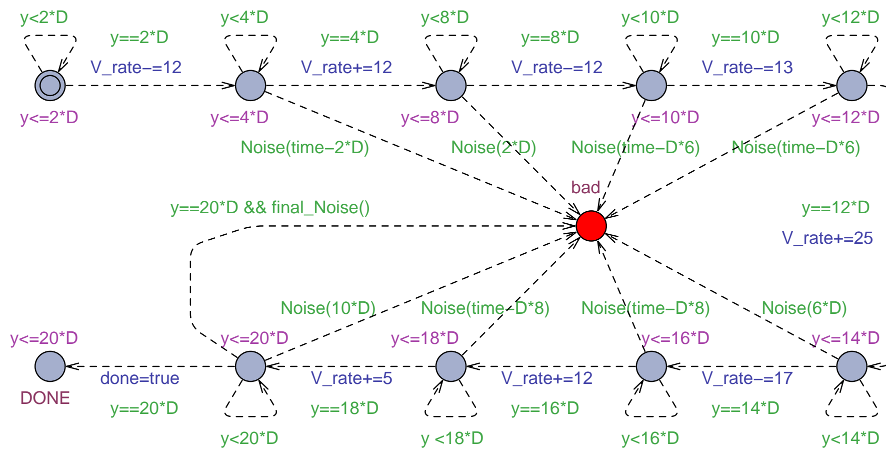
We use a third automaton represented on Fig. 9(c) to schedule the composition. Initially it sets the value of the volume to V0 and then it repeats the following actions: it first updates the global variables V, V_acc and time through function update_val. Then the scheduling is performed using the two channels update_cy[4] and update_pump. When the end of the cycle of the machine is reached, the corresponding automaton sets the boolean variable done to true, which forces the scheduler to go to location END.

Simulation with SIMULINK. Fig. 10 shows the SIMULINK block diagram for simulation of the strategies synthesised by UPPAAL-TIGA. The diagram consist of built-in functions and four subsystems: Consumer, Accumulator, Cycle timer and Pump activation (we omit the details of the subsystems). The Consumer subsystem defines the flow rates used by the machine with the addition of noise: here the choice of a uniform distribution on the interval $[-\epsilon, +\epsilon]$ with $\epsilon = 0.1l/s$ has been made. The Accumulator subsystem implements the continuous dynamics of the accumulator with a specified initial volume ($8.3l$ for the simulations). In order to use the synthesised strategies the volume is scaled with a factor 10, then rounded and feed into a zero-order hold function with a sample time of 20s. This ensures that the volume is kept constant

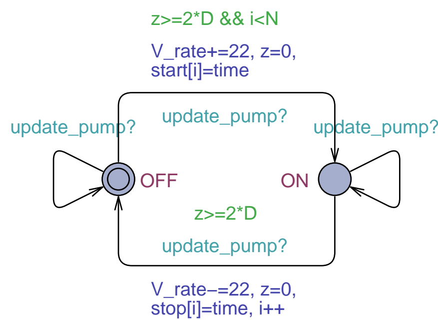
[2] For robustness, we restrain safety constraints of $0.1\ l$.

[3] Notice that we impose a bit more than P_1 as we require that 2 seconds have elapsed at the beginning of the cycle before switching on the pump.

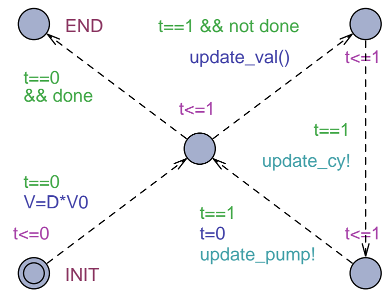
[4] We did not represent this synchronization on Fig. 9(a) to ease the reading.



(a) Model of the cyclic consumption of the machine



(b) Model of the pump



(c) Model of the scheduler

Figure 9: UPPAAL-TIGA models.

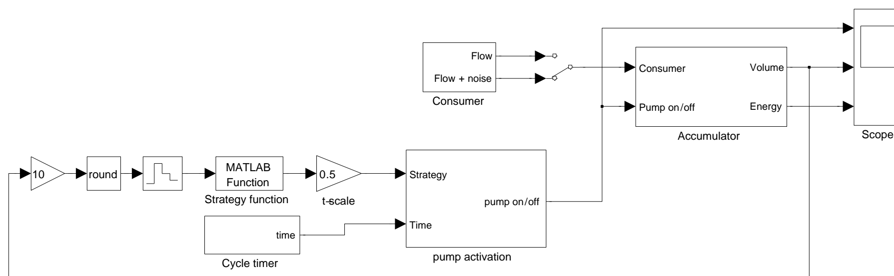


Figure 10: The overall SIMULINK model.

during each cycle, which is feed into the strategy function. The Pump activation subsystem takes as input the on/off dates from the strategy (for the given input volume of the current cycle) and a Cycle timer, that holds the current time for each cycle.

2.2.3 First Results

The results of our modelling/synthesis/verification/simulation methodology show that the controller synthesised with UPPAAL-TIGA is robust whereas the robustness of the Smart controller is unsettled. More interestingly, the simulation reveals that the performances of the synthesised controllers provide a vast improvement both of the Smart Controller (33%) and of the 2-point Controller (45%). The methodology and results are reported details in [5].

3 CHESS: Wireless Sensing

For a description of this case study, we refer to [7].

3.1 ESI/RU

3.1.1 Participants

- Faranak Heydarian, Radboud University;
- Julien Schmaltz ESI/RU Eindhoven;
- Frits Vaandrager, Radboud University.

We consider the formal analysis of the timing parameters of the gMAC protocol developed by CHESS for Wireless Sensor Networks (WSN). Each node of a WSN has its own hardware clock, which is drifting apart from the other clocks. The protocol implemented by CHESS (1) includes a clock synchronisation mechanism, and (2) a "guard" timed used to delay the sending of messages so that even the slowest nodes are ready to receive. The guard time is key to the functional correctness of the WSN, but also to its energy consumption. Our goal is to analyse the functional correctness of CHESS solution with respect to the value of the guard time.

3.1.2 Model

We model CHESS solution as a network of timed automata which is analysed using the UPPAAL model checker. Every node i is composed of three automata: the clock, the wireless sensor node, and the synchroniser. Each automaton is parameterised by index i. The network is represented by the parallel composition of all automata of all nodes. Table 1 presents the main parameters of our model.

Parameter	Description	Constraints
N	number of nodes	$0 < N$
C	number of slots in a time frame	$0 < C$
n	number of active slots in a frame	$0 < n \leq C$
$Nodes$	set of nodes	$Nodes = [0..N-1]$
$tsn[]$	array of TX slot numbers	$\forall i \in Nodes, 0 \leq tsn[i] < n$
		$\forall i, j \in Nodes, i = j$ iff $tsn[i] = tsn[j]$
k_0	number of clock ticks in a time slot	$0 < k_0$
g	guard time	$0 < g, g+1 < k_0, 2.g < k_0$
min	The minimal time between two clock ticks	$0 < min$
max	The maximal time between two clock ticks	$min < max$

Table 1: Network Parameters

The clock. The hardware clock is modelled as the timed automaton depicted in Figure 11. The clock drift is represented by the constants min and max. A clock tick is produced in at least min time units and before max time units. This model also includes jitter as the difference between ticks is not constant over time.

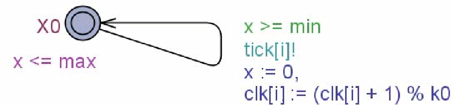


Figure 11: Timed automaton of a wireless sensor node's clock

This automaton has the following behaviour. It performs a synchronisation action $tick[i]!$ when its clock variable x has reached a value between min and max time units, and then returns to its initial state by resetting x to zero. With each tick, the clock updates the value of $clk[i]$ by $(clk[i] + 1) \bmod k_0$ to refresh the position of the pointer of the current slot. The length of a slot is k_0 clock cycle.

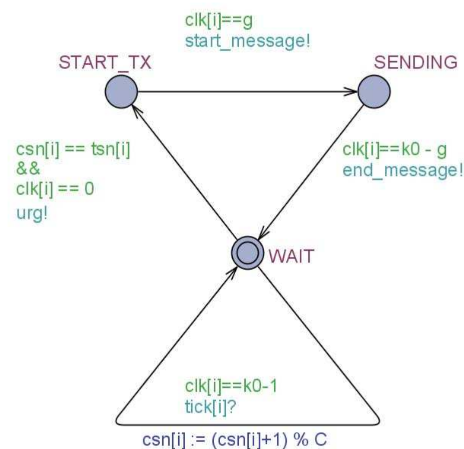


Figure 12: Timed automaton for a single wireless sensor node

The wireless sensor node. Initially, the node is in location "WAIT". At the beginning of a slot (i.e., when $clk = 0$), if the current slot number (csn) equals the transmission slot number (tsn), node i is transmitting and moves to location "START_TX". The node waits for the guard time (g) before moving to location "SENDING" where it effectively sends a message. The transmission ends g time units before the end of slot, i.e., when $clk = k_0 - g$. The automaton moves then to the location "WAIT" and wait until the end of the slot (i.e., when $clk = k_0 - 1$) to increment its current slot number on the following clock tick. If at the beginning of a slot, a node is not sending, it stays in location "WAIT" for the duration of the slot. At the end of the slot, it increments its current slot number.

The synchroniser. This automaton implements the clock synchronisation mechanism of CHESS. In our model, this synchronisation happens on the transition from "S1" to "S0" (see Figure 13). Counter clk is reset to $g+1$. Each node is expected to receive a message after g (where g denotes the value of the guard time) clock cycles. We add 1 to take into account that resetting the counter takes one additional cycle.

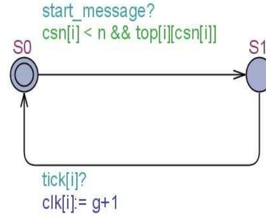


Figure 13: Timed automaton of a synchroniser

Synchronisation happens when a node starts sending a message, i.e., when producing action $start_message$. The topology is represented by Boolean matrix top. Each element $top[i][j]$ represents the existence of a connection between node i and the node transmitting in slot j, i.e. the node with index $tsn[j]$. Node i is allowed to synchronise if it has a connection with the node transmitting in the current slot, i.e., if $top[i][csn[i]]$ is true. Using this modelling feature we can easily modify the topology and the neighbour relationship by simply modifying the values in top. For instance, we can easily take into account that when a node is sending, it cannot receive any message. Let node i be the node sending in slot j. Then, by simply setting $top[i][j] = false$, node i does not synchronise when it is sending. In a similar way, the neighbour relationship may also be asymmetric, i.e., node i can send a message to node k, i.e., $top[k][j] = true$, but node i does not receive messages from k. If node k is transmitting in slot l, then we set $top[i][l] = false$.

3.1.3 First results

We formalised using temporal logic the precise meaning of being synchronised. Informally, nodes are synchronised if they are all in the same slot when a message is sent. We modelled different topologies and used UPPAAL to obtain the smallest value of g which guarantees synchronisation.

From experimental results, we derived a general formula and proved it a necessary condition to achieve synchronisation for a *fully connected* network, i.e., all nodes are connected to each other. This formula gives the minimum value of g as a function of the topology, the drift and jitter of the clocks, and the allocation of slots. This formula shows that – for a given drift and jitter – the minimal guard time decreases when the number of nodes increases. Intuitively, if a node receives more messages in a frame, it synchronises also more often. Hence, it tolerates a smaller guard time.

We extended our model to support topologies where nodes are not fully connected. Experimental results have shown that the formula for fully connected topologies cannot be generalised to non-fully connected ones. Our experiments also show that the value of g increases when the

number of nodes increases. Intuitively, when nodes are not fully connected, a node receives less message in each frame. Therefore, it synchronises less often and the guard time must be increased.

Despite the limited size of the networks we were able to analysed (up to 5 nodes), our models and their analysis resulted in a better understanding of the protocol, and more insight for us and our industrial partner. Several meetings were held at CHESS to ensure that our models were adequate with respect to CHESS implementation. In the future, we plan to make our model even more realistic by adding collisions and dynamic topologies, i.e., nodes which join and leave the network. We are also studying possible abstractions that would enable the analysis of larger networks.

All our results are available in a recent technical report of the Radboud University Nijmegen [8].

3.2 RWTH Aachen

3.2.1 Participants

- Haidi Yue,
- Henrik Bohnenkamp,
- Joost-Pieter Katoen,

all RWTH Aachen University.

3.2.2 Model

We consider the gMAC protocol developed by Chess for Wireless Sensor Networks. In this protocol, a Time Division Multiple Access(TDMA) scheduling is employed to allow multiple sensor nodes to share the same transmission medium. Collisions happen if two or more nodes send messages to another node at the same time. Due to the energy limitation, the number of active slots S_A in *TDMA* should be chosen as small as possible. However, a smaller S_A yields greater probability of collisions.

To investigate the relation between the number of active slots, collisions, and energy consumption, we model the gMAC protocol with MoDeST, a modelling language developed by University Twente, RWTH Aachen University, and Saarland University [3]. Our means of analysis is discrete-even simulation with Motor/Möbius [2, 4].

The objective of our analysis is to gain insight into the behaviour of the gMAC protocol with a larger number of nodes in a network. In particular, we are interested in the mechanism for collision detection using piggyback information, the numbers of collisions detected and undetected, and the influence of the number of active slots on the the energy consumption per node.

Mobility and clock drifts are currently not considered. The network is a static 15×15 grid with fixed neighbourhood relation, which depends on the transmission range of the nodes radio. Every node is assumed to have the same transmission range. We assume transmissions ranges with at most 4 and at most 8 immediate neighbours.

In the following we will describe the key components of the MoDeST model. A node has a unique number as identifier `id`, and a clock `c`, which measures the length of a slot and is then reset to 0 to measure the next slot, and so forth. Furthermore, a node maintains its view as an array `view` of booleans of length equal to the number of active slots.

Communication between nodes is modelled by means of a global array of buffers, where a node accesses its buffer with index `id`, `buf[id]`. A buffer contains two variables `is_written_to` and `writers` (used as counting semaphores) to coordinate senders and receivers and to detect collisions. If sending is successful, a buffer contains the piggy-back information for collision detection. A similar array `energy` is used to keep track of the energy consumption.

The behaviour is modelled by using extensive data-manipulation. In particular, variables are updated with statements in `{= ... =}` pairs, and a node executing a `{= ... =}` block does so atomically.

```
// macro SEND()
// we assume c == 0 here
when (c == guard_time)  // guard_time is over
  start_sending {=
    energy[id] += 0.001*PL_TRANSMITTING, // add energy spent on sending
    for all neighbours:
      bufs[neighbour].is_written_to += 1
      bufs[neighbour].writers += 1
  =};
when (c == slot_length - guard_time) // end of send period
  msg_sent {=
    for all  neighbours:
      bufs[neighbour].writers -= 1,
      if(bufs[neighbour].is_written_to == 1){ // != 1 means collision
        // copy view array to buf
      }
  =};
when (c == slot_length) // end of slot
  reset_channel  {=
    // increase slot number
    c = 0,
    for all  neighbours:
      // set everything in bufs[neighbour] to 0
  =}
```

Figure 14: Mechanism for sending

```
// macro RECEIVE()
when(bufs[id].is_written_to == 1 // exactly one process wrote
     && bufs[id].writers == 0 // the one sender is done with writing to buffer
     && c >= slot_length - guard_time ) // end of send period
  msg_received    {=
    view[slot_nr] = 1,         // we received smthg in slot slot_nr
      if (pigyback-information indicates collision )
         // choose new send slot randomly
  =}
```

Figure 15: Mechanism for receiving

In Figure 14 we see a sketch of the send operation as modelled in MoDeST. Before a node actually sends in its send slot, it must wait until the guard time has passed and the send period begins. In the send period and the time until the next slot starts, three steps are executed, which are marked with the three action names `start_sending`, `msg_sent`, and `reset_channel`. Each of these actions is accompanied by a {= =} block, where the actual work is done. For `start_sending` this means to increase counters `is_written_to` and `writers` in all buffers of the neighbour nodes. Then, when the send period is over, action `msg_sent` is executed. The node decreases in all neighbours buffers counter `writers` and checks if a collision has occurred (this is what we call a *real collision*). If no, the senders view copied to the neighbours buffer. Finally, at the end of the send slot, with action `reset_channel`, the neighbours buffers are cleaned up and the next slot is prepared.

In Figure 15, we see the step that are executed if a message is received by a node. This action is executed at the end of the send period of the sending nodes, and only if the counter in the node's buffer is equal to 1. In that case, the piggy-backed information of the sender is checked

to detect a collision. If a collision is detected (this is what we call a *detected collision*), a new send-slot is chosen probabilistically.

Before a node can receive, it has to listen first. This is modelled as sketched in Figure 16. Note the choice modelled by the `alt` construct: `RECEIVE()` stands as a placeholder for the fragment in Figure 15. We thus have a choice between the timed guard `c == slot_length` and the untimed guard of the `RECEIVE()` process. If `c == slot_length` becomes true, nothing has been received in the slot, and the next slot is prepared.

Note that the coordination between processes is not done via synchronising actions (the shown actions are all hidden in the parallel composition), but only by means of shared variables. This is necessary in order to be later able to extend the model with clock drifts and jitter.

The described behaviour is put together in process `Node`, and all the Nodes are put in a parallel composition, as sketched in Figure 17.

```
// macro LISTEN()
when ( current_slot != my_send_slot ) // we are listening
listen {=
    energy[id] += 0.001*PL_RECEIVING
  =};
  alt {
    :: when(c == slot_length)
       {=
         view[slot_nr] = 0   // nothing received
        =}
    :: RECEIVE() /* as explained above */
  };
  when (c == slot_length)
  {=
    // increase slot number,
    c = 0
  =}
```

Figure 16: Listening for incoming message

3.2.3 First Results

We have conducted several simulation experiments, where we estimated, first, the effectiveness of the collision detection mechanism, and second, a rough measure for the energy consumption vs. the number of active slots.

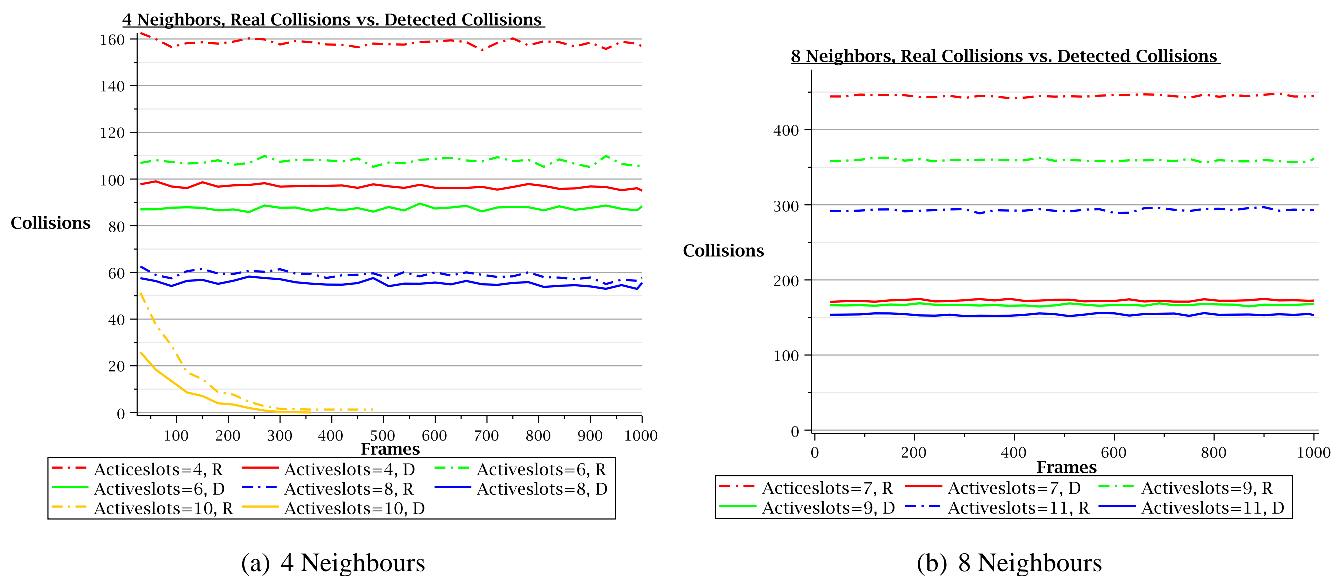
In Figures 18 we see the evolution of the number of collisions (y-axis) for the current frame (x-axis). Figure 18(a) shows the case for 4 neighbours within the transmission range, and Figure 18(b) the case for 8 neighbours. The graphs show that the number of active slots does influence the number of collisions enormously, but that very many active slots are needed (compared to the number of neighbours) before the number of collisions actually tends to go to 0. In the case of 8 neighbours, this is the case from 23 active slots upwards (not shown in the graph).

In Figures 19, we see the total energy of a node with 4 neighbours that is needed to transmit 50000/100000 messages successfully. Although a larger number of active slots speeds up the process (fewer frames are needed), also the energy consumption increases. The optimum seems to be 4 active slots for the 4-neighbour case, and 7 active slots for the 8-neighbour case.

```
process Node(int id) {
  // initialisations
  do {  // repeat for each slot
    :: when (slot_nr > activeslots) ... // idle until frame ends
    :: SEND()...
    :: LISTEN()...
  }
}

par{
  :: hide all actions in (Node(1))
  :: hide all actions in (Node(2))
   ...
}
```

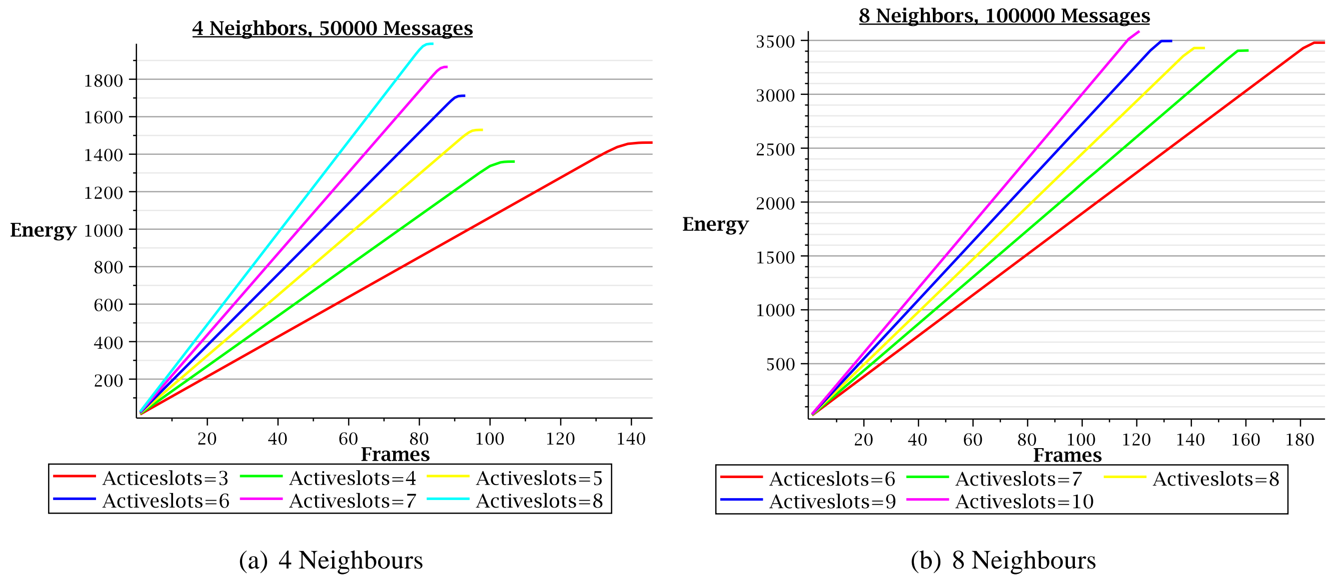
Figure 17: Rough structure of process `Node` and parallel composition



(a) 4 Neighbours

(b) 8 Neighbours

Figure 18: Collisions, real and detected



(a) 4 Neighbours (b) 8 Neighbours

Figure 19: Energy consumption

In the future we plan to extend the model and the analysis in several directions.

- Developing different criteria on which to base the assessments of energy efficiency;
- considering mobility;
- considering clock drifts;
- finding analytical explanations for simulation results.

Bibliography

[1] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-TiGA: Time for playing games! In *19th Int. Conf. CAV 2007*, volume 4590 of *LNCS*, pages 121–125. Springer, 2007.

[2] H. Bohnenkamp, T. Courtney, D. Daly, S. Derisavi, H. Hermanns, J.-P. Katoen, V. Vi Lam, and W. Sanders. On integrating the Möbius and MoDeST modeling tools. In *Proc. DSN 2003*. IEEE CS Press, June 2003.

[3] H. Bohnenkamp, P. R. D'Argenio, H. Hermanns, and J.-P. Katoen. MODEST: A compositional modeling formalism for hard and softly timed systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.

[4] H. Bohnenkamp, H. Hermanns, and J.-P. Katoen. Motor: The MoDeST tool environment. In *Proc. TACAS'07*, volume 4424 of *LNCS*, pages 500–504. Springer-Verlag, 2007.

[5] F. Cassez, J. J. Jessen, K. G. Larsen, J.-F. Raskin and P.-A. Reynier Automatic Synthesis of Robust and Optimal Controllers – An Industrial Case Study. Hybrid Systems Computation and Control (HSCC'09), forthcoming, 2009.

[6] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2), December 1997.

[7] Holger Hermanns, Poul Hougaard, Teun van Kuppeveld, Kai Sven Mittermller, Jan Storbank Pedersen, Marcel Verhoef, Ivo van Vessem. Quasimodo Deliverable D5.2. *Preliminary descriptions of case studies.*. July, 2008.

[8] F. Heydarian, J. Schmaltz, and F.W. Vaandrager. Formal verification of clock synchronization of a simple wireless sensor network. Technical report, Radboud University Nijmegen, 2008. In preparation.

[9] Kai Mittermüller. Modellierung einer Speicherladeschaltung. Bachelor's thesis. Saarland University, 2008.

[10] Simulink, 2008. `http://www.mathworks.com/products/simulink/`.