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

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

Project Acronym: **QUASIMODO**

Deliverable no.: **D 5.2**

Title of the deliverable: **Preliminary descriptions of case studies**

Contractual Date of Delivery to the CEC:	**Month 6**
Actual Date of Delivery to the CEC:	**Month 12 (February 1, 2009)**
Organisation name of lead contractor for this deliverable:	**Saarland University**
Author(s):	**Holger Hermanns, Poul Hougaard, Teun van Kuppeveld, Kai Sven Mittermüller, Jan Storbank Pedersen, Marcel Verhoef, Ivo van Vessem**
Participants(s):	**P05 SU, P07 TERMA, P08 CHESS, P10 HYDAC**
Work package contributing to the deliverable:	**WP5**
Nature:	**R**
Version:	**2.0**
Total number of pages:	**27**
Start date of project:	1st January 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 provides an overview on the case studies pursued in the context of Quasimodo.

Table of Contents

1 Introduction

The Quasimodo project plans to carry out a series of challenging case studies, provided by our industrial partners in which related families of models are used for (quantitative) analysis, code generation and test generation. In order to demonstrate and challenge the usefulness of our methods and tools, and assess their strengths and weaknesses, it is important to apply them on realistic problems. Therefore we have selected case studies that are very close in spirit to products that are currently under development by our industrial partners. Our expectation is that this is highly motivating both for the academic and industrial partners.

This deliverable provides preliminary descriptions of the case studies we plan to cover. These are

- HYDAC: Accumulator-Charge Controller
- CHESS: Self-balancing scooter
- CHESS: Wireless Sensing
- TERMA: Herschel/Planck

2 HYDAC: Accumulator Charge Controller

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 optimizes the energy and the wear of the used components, especially the pump.

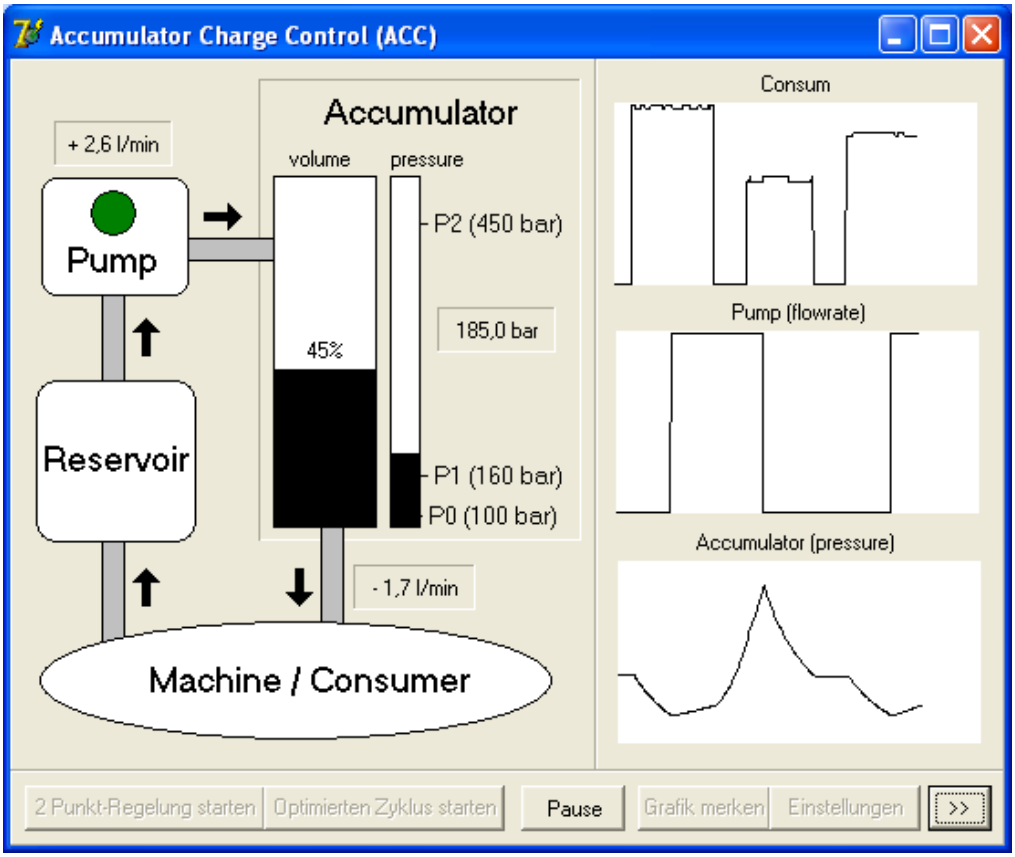


Figure 1: Screenshot from the program which simulates the ACC

- A hydraulic system has 4 basic components (Figure 1):
- Reservoir: The tank contains the oil which is used by the system.
- Pump: The pump pumps the oil from the tank to the hydro-pneumatic accumulator which stores the pressure.
- Machine: The machine operates in cycles from 10s to 5min. In each cycle the machine needs a characteristic amount of pressure and oil. This amount includes several consumers, like e.g. cylinder.
- Hydro-pneumatic accumulator: Fluids are practically impossible to compress and can therefore not store any pressure energy. In hydro-pneumatic accumulators, the compressibility of a gas is used to store fluid. The gas is in the most cases nitrogen. The accumulators consist of part fluid and gas with a bladder or diaphragm as separating element (see Figure 2, 3). Only the fluid chamber is connected to the hydraulic circuit. When the fluid pressure rises, the gas is compressed, when pressure falls, the compressed gas expands and forces the accumulated fluid to flow into the cycle.

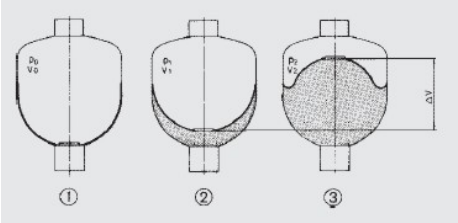


Figure 2: diaphragm accumulator

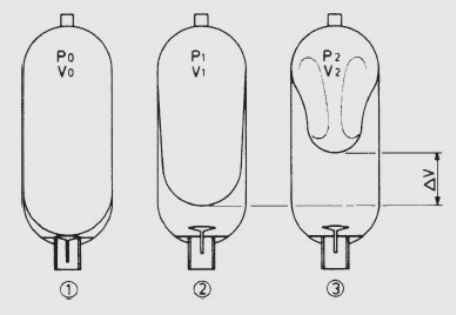


Figure 3: bladder accumulator

(1) The accumulator is pre-charged with nitrogen.
(2) The accumulator is charged with the minimum pressure for operating.
(3) The accumulator is maximal filled with oil. The maximum pressure is reached.

The interesting value of the accumulator is the volume of oil stored, because the accumulator is so dimensioned that the required pressure can always be provided. It is obviously, that the volume of the oil is $V_{oil} = V_{accumulator} - V_{gas}$. In the following, we will always talk from the volume of the gas, because this volume can be calculated by the gas equation:

$$P * V = const. * T$$
P=pressure, V=volume, T=temperature

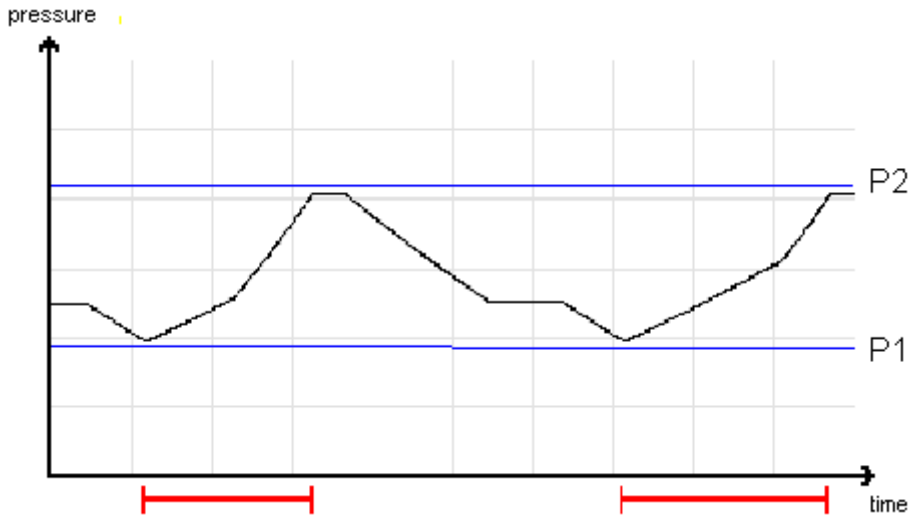
The problem by the gas equation is, that in the system because of physical restrictions (high pressure and very fast changes) and costs, only the pressure can be measured.
There are two main simplifies of the gas equation:

1) Isotherm: T = const. => $P * V = const.$
2) Adiabatic: $P * V^x = const.$ x = adiabatic exponent (depends on many factors)

In most cases, the changes of state tend to follow adiabatic rather than the isothermal laws. It is often the case that the charge takes place isothermally and the discharge adiabatically. In practice a good approach is $P * V^{1.3}$. We'll using this approach for our ACC too.

2.1 State of the art

The ACC refills the accumulator by two pressure points P1 and P2 in which the pump get started and stopped.



This design is very simple, but not optimal. The pump control is independent from the consum of the machine, so there must be always reserves for the worst case consumptions. This yields to a high wear and energy usage.

2.2 Improved design

The main goal for the new ACC is to reduce the pressure in the accumulator by refilling it intelligent. This yields to less energy use (in average ~40%) and because of lower pressure to less wear by the pump and the bladder/diaphragm. For further improvement of the pump wear we take care of minimum run- and downtime of the pump, given as parameter in the ACC.

First the ACC run a cycle with two-point controller like described above and measure the pressure and the time the pump is running (~ all 10 ms, depending on the duration of the cycle with a precision of 12bit). The energy needed, which should be optimized, is defined as

$$E = \int P(t) - P0\, dt \quad , where\ P(t) = \text{measured pressure at time t}$$

$$\text{and P0} = \text{pre} - \text{charge pressure}$$

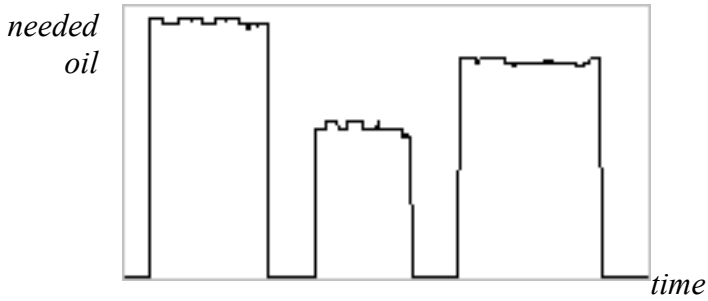
The consumer of the machine will be identified by calculate

$$C(t) = (V(t) - PuSt(t) * pumppower)'$$

$$V(t) = calculated\ \ from\ P(t)\ with\ \ the\ gas\ equation$$

$$PuSt(t) = \begin{cases} 1, & pump\ is\ running \\ 0, & pump\ is\ stopped \end{cases}$$

This leads to a curve, that looks like the following (with some kind of smoothing),



which defines the characteristic consumption of the machine. The consumption is different from machine to machine.
Based on this consumption, which is provided from C(t), the new “intelligent” time-points for switching the pump are calculated.

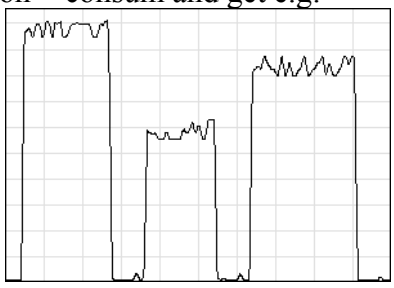
2.3 Description of the ACC-Algorithm

As described before, the ACC stores the pressure and the pump state during the cycle. On the fly it is possible to calculate the volume curve without pump from this values. Now we want to have a look at the optimization algorithm. The algorithm is designed to be very fast does not use dynamic programming.

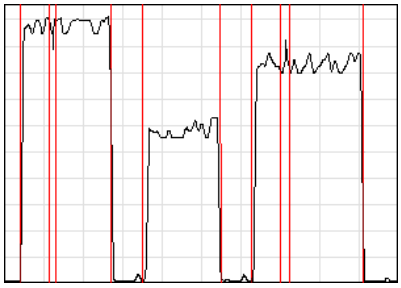
```
void optimize_acc(MYFLOAT CurrentPressure) {
    int i;

    //smoothing of the curve
    FilterValues(10);
    //derivation => consumption
    for (i=StoreCount-1;i>0;i--)
        Values[i]=(Values[i-1]-Values[i])*Rate;
    Values[0]=Values[1];
    //detection of single consumers
    SearchAreas(0.4, 40);
    //calculation of corresponding pump online time
    CalcPumpIntervalls();
    //removes short pump runtimes
    EliminatePeaks(20);
    //calculate maximal energy optimum
    FitPumpSPsToOptMinPressure(CurrentPressure);
    //respect Minimum stop- and runtime
    CalcMinPumpStoptime();
    CalcMinPumpRuntime();
}
```

Figure 4: sequence of optimization

FilterValues: The consumption curve is the derivation of the the volume curve without pump. Hence through derivation the noise and the artifacts of the discretization increase, the noise has to be removed before derivation. This is done by calculating the average of 10 values.
After that we calculate the derivation = consum and get e.g:



SearchAreas: Now we try to identify every single consumers. That means we want to identify the intervals where a consumer is active. So we get something like the lower picture . The borders of the areas are marked by the red lines. For each area we calculate the average consumption. After this step, we don't need the curve any longer, but only work with the area intervals and the corresponding average consumption.



CalcPumpIntervals: In this step we transform the areas with the average consumption to the corresponding pump-runtimes. That means, how long the pump has to run to balance the consum. Hence after this step we only have areas of pump runtime. Please remark, that it is possible that the intersection of two intervals is not empty.

EliminatePeaks: Now we remove very short pump runtimes,which are created through small areas or areas with no consumer, and part of runtimes outside the cycle. The pump runtime of this areas will be added to the previous area.

FitPumpSPsToOptMinPressure:
After a cycle with 2-point-controller the pressure can be much higher or lower, than it would be necessary for an optimal cycle. So in this function we increase or decrease the first pump runtime intervals, such that the pressure reaches an optimal value.

CalcMinPumpStoptime:
Adjust the pump runtimes, to hold the minimum pump runtime.

CalcMinPumpRuntime:
Adjust the pump runtimes, to hold the minimum pump stoptime.

2.4 Research Questions

The following questions are supposed to guide modelling and analysis of this case within Quasimodo.

- Is our controller safe in the sense that the pressure is always in a safe region?
- Is the controller optimal under the limitations of runtime and downtime of the pump?
- Is the controller robust to fluctuations in the cycles?
- How much does the improved solution differ from the optimal one, as a consequence of the simplifications we have made?
    - adiabatic gas equation
    - constant pump-power (in real depends on the pressure)
    - long-run drift of the systems characteristics e.g through leakage.

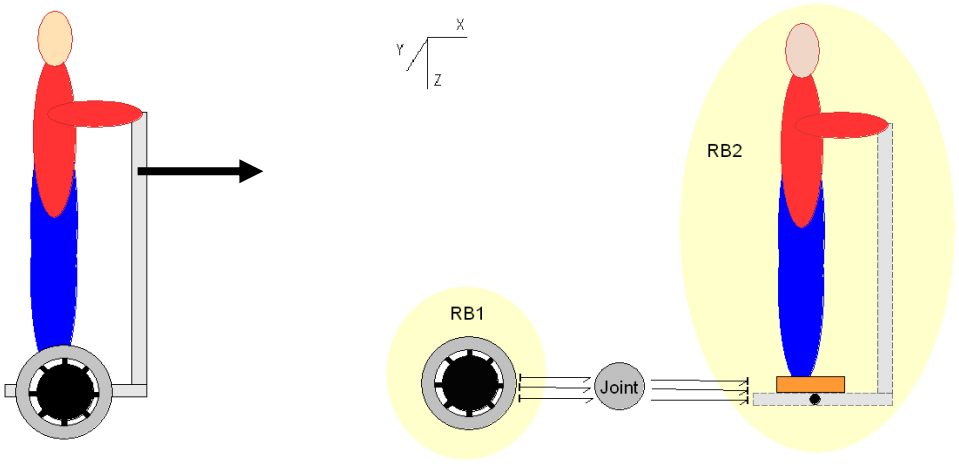
3 CHESS: Self-balancing scooter

This section describes a project where motion control is an essential part and where problems can arise due to improper software design. Improper design is for example a result of the misunderstanding due to different views between the engineering disciplines involved.

This case study is proposed by Chess and has already been used to validate formal development approaches. It is the so-called self-balancing scooter, displayed in the figure on the right. The self-balancing scooter is chosen because it is a meta-stable system with interesting control challenges. It exposes design problems in all engineering disciplines involved: mechanics, electronics and software development. An inappropriate development method would easily result in a non-functional or badly performing system. During this case study, the model driven development approach is tested with the use of the trajectory. A dynamic model of the self-balancing scooter is introduced which gave the opportunity to develop and verify controllers and test the properties of the electronics even before the hardware was available. In parallel with the development of the model and control laws, the hardware of the system is developed. The hardware is developed with the use of commercial of the shelf (COTS) products, based on FPGA technology and the physical model is developed in the 20-sim tool.

3.1 Dynamic model

As previously stated, a dynamic model of the system is developed during the project and extended to provide correct information during each different development stage. In line with the method a first model of the physical system is made. The system is divided into two rigid bodies which are represented with the use of bondgraphs, see the figure below.



Verification is done with common understanding about the dynamics of the application. For example, a self-balancing scooter will fall down when a small deviation from upright position is the initial condition. The verification is done with the use of a representation with graphs and a 3D animation.

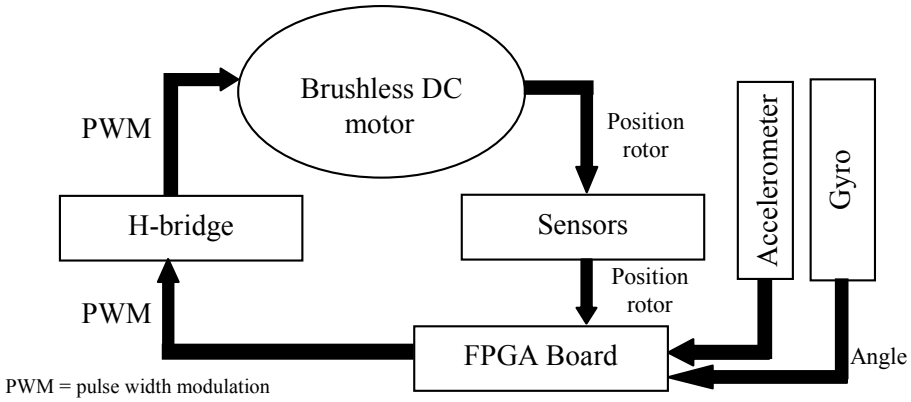
The control law design will be done with a linearized representation of the dynamic model. With the linearized model the poles and zeros are plotted. The pole and zero plot gives information about stability and control possibilities of the system. During the design of the control laws various design choices are investigated, for example which sensors need to be used. We have investigated the use of a gyroscope and an accelerometer. Verification is done with a model with a deviation of the upright position which returned to the upright position and a model with a simulated centre of mass movement which showed a forward movement of the self-balancing scooter. Simulations showed a better stability with the use of a gyroscope.

Implementation of the embedded system properties gives a discrete model. A discrete model requires a redesign and recalculation of the control law properties since the discretization introduces phenomena that are due to discrete sampling. Verification of this model is done with the initial condition and forward movement test also used with the control law step. When the verification results are satisfying the software code can be generated from the model.

Next step is the realization with the integration of the various disciplines into a working system.

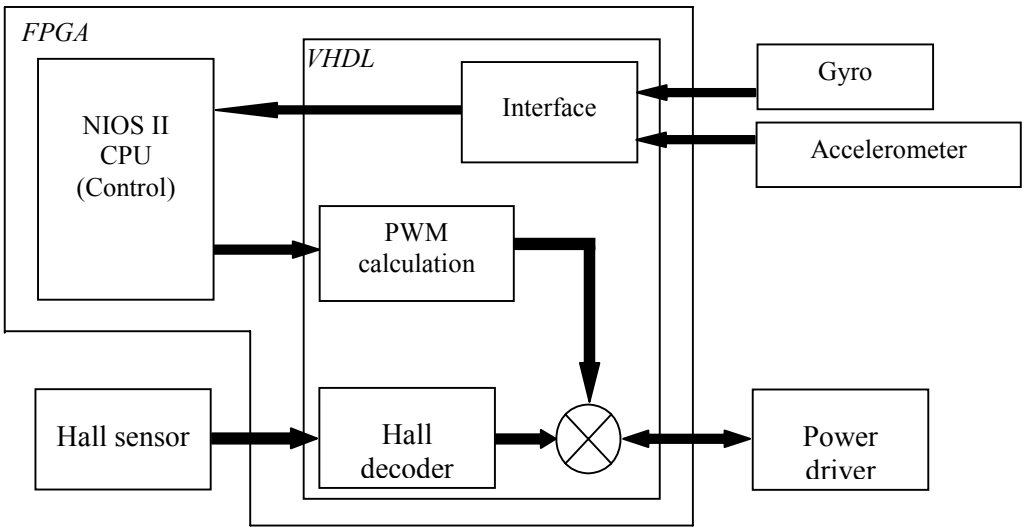
3.1.1 Electronics

For the development of the electronics the following high-level system architecture is used.



The H-bridge transforms the PWM from the FPGA to the correct power level for the motor. It is basically a clever signal amplifier. The position of the motor is measured with the use of van Hall sensors. With the use of these sensors the motor can be controlled correctly, but it has a low resolution. The accelerometer and gyro are used to measure the upright position of the self-balancing scooter. At this moment the accelerometer is used for correcting the drift of the gyro. The last block is the FPGA board, on which the software will be executed.

The choice during the design was the use of an FPGA (Field Programmable Gate Array) to control the system. With the use of an FPGA great opportunities arise in the form of flexibility in the design and in reuse of standard intellectual property (IP) blocks. Besides the IP for the FPGA the designed controller board should also be made with several possible additions in mind. For the design of the prototype electronics commercial off the shelf development boards were used to reduce design time and costs. With these choices a hardware platform was developed which can be used in different applications.



As previously noted the FPGA board is used to execute the software. Software can be executed after the creation of a VHDL (VHSIC hardware description language) platform. An overview of this platform is given above. In the figure it is shown that the platform consists of a VHDL part and

a soft core called NIOS II (the processor). Besides the FPGA description the inputs and outputs are displayed in the figure. With the use of the NIOS II core it is possible to execute ANSI C-code and with the use of the VHDL blocks it is able to interface to the hardware.

3.1.2 Integration

With the completion of each different discipline integration can take place. After integration the application was tested and validated with respect to the requirements. With the resulting system it is possible to move forward and backward and stand in upright position on it.

3.2 Challenges

During the project a few challenges were encountered, without the possibility to use a method to solve them. A new approach is needed to tackle these challenges. The introduction of a new methodology could result in the reduction of the parameters/functions that need to be defined by trial and error or by previous experience. This reduction will prevent the extensive risk on design time and costs due to unknown design issues. Key issues are energy efficiency and safety.

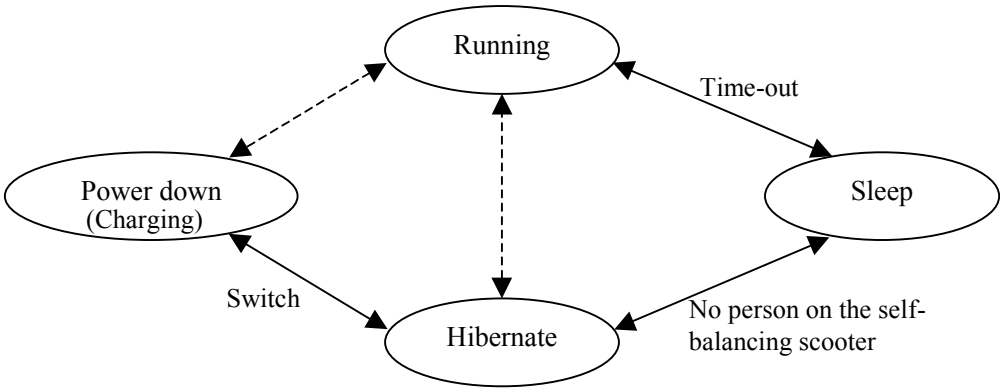
3.2.1 Energy efficiency

The motion control solutions developed by Chess are used in mobile systems. The mobile character requires more effort on the design of the system. For mechanics and electronics this can be seen in the use of light weight materials, the development of more efficient motors and the use of low power electronics. Besides the mechanics and the electronics the software could also contribute to a more efficient system.

Higher energy efficiency can be realized for example by a lower execution frequency of software or a better software management of peripherals use. The management of peripherals use can be done with the use of formally defined system states. As previously described, the system state definition is dependent on the choice of the software architect or engineer. With the introduction of a formal method for the definition of the system states a better result is possible. Besides the definition of the system states a correct definition of the transition conditions is needed.

During the test case of the self-balancing scooter a good definition of the system states was missing. This resulted in lower energy efficiency and less structure in the system states. A redesign is needed, this time with a definition of the systems states. A first definition is displayed in the figure below and a more detailed description is given in the table. The dotted lines in the figure are transitions that may also be needed.

The challenge for the Quasimodo project is to come with a formal definition of the system states and the needed transition conditions. This results in a correct working system with better energy efficiency. Management of peripherals use within the system states requires attention to the interaction between the user and the resulting solution. A perfect solution will result in better energy efficiency without the user noticing any difference. For example switching off unused peripherals requires a very fast wake-up procedure or the use of extra sensors.



Component versus Operating mode	Running	Sleep	Hibernate	Power down
FPGA	ON	ON	OFF	OFF
CLK FPGA	ON	OFF	OFF	OFF
RAM	ON	ON	OFF	OFF
External IO	ON	ON	OFF	OFF
Motor	ON	ON	OFF	OFF
Battery Monitor	ON	ON	ON	ON
Power management control	ON	ON	ON	OFF
Motor safety switch	ON	OFF	OFF	OFF

For the definition of the transition conditions information of the self-balancing scooter may be needed, sensor information that is needed can be defined. With the current implementation, not all information may be available, however this can be implemented.

3.2.2 Safety

Motion controllers that are used in products where close interaction with humans is involved require extra safety criteria. Improving the safety of the system is possible by creating redundancy of the used components. Redundancy guarantees the correct working of a system with the use of multiple components for the same function. Besides the redundancy, software can also improve the safety of the system. With a correct definition of the system states and transitions it becomes possible to create a safer system. Besides transition conditions a definition of the failure classifications needs to be made. The failure can be classified with fatal, error and warning. Each failure will have a result, fatal could be system halt and warning could be a flashing light. This classification can be used for the concepts graceful degradation and fail-safe termination. Graceful degradation can be used when an exception is detected and a part of the system is still able to operate. Fail-safe termination makes sure that the system stops in an appropriate way without causing a dangerous situation.

- Exceptions in a motion system which can have an influence on the safety are for example:
- Signal out of range (motor current, angle).
- Battery conditions (overheated, low, empty).
- Malfunctioning components, electrical or mechanical.
- Exceptions in software.

For a correct definition of the states and the transition conditions the exceptions of the system need to be considered.

With the design of a self-balancing scooter safety measures are also needed. Without the correct safety measures the person using the product could easily get hurt. For example when an exception is detected in the software and an error state is activated this could lead to a definite shutdown of the system. When the system is in use and the controller stops due to an exception this could lead to dangerous situations. A good definition of transitions and states is needed. Previously proposed solutions resulted in an exploded number of system states and transition conditions, without the possibility of verification.

The challenge for the Quasimodo project is to come with a formal definition of system states and transition conditions providing graceful degradation and fail-safe termination.

3.3 Research Questions

The definition of system states and state transitions forms a great risk in the development of a mechatronic control system. The use of a method and tools possibly reduces this risk. The tools and methods need to support the development of a verifiable system state model, which results in the optimum solution. Besides the verification of the system states the model can also be used as an input for a code generation mechanism. Code generation can be used for the reduction of development time. A model can also be used for the definition or generation of the test specifications.

- Prepare a model to investigate the system behavior.
- Evaluate the behavior for all state transitions.
- Find optimal behavior: Safe and user friendly.
- Generate code from the model.
- Generate test cases from the model.

4 CHESS: Wireless Sensing

Wireless Sensor Networks are envisaged to consist of thousands of autonomous sensor nodes. Sensors communicate wirelessly with neighboring sensors thus forming a sensor network.

Sensors rely on these neighbours to forward their messages such that these messages eventually reach their destination where they can be processed. In this scenario the financial budget per sensor is severely limited, which translates to a limited amount of chip area and a low capacity energy supply. This in turn puts tight restrictions on the amount of available storage, the affordable computational complexity, the amount of data that can be transmitted, and the transmission range. Medium access control protocols for such networks are in the focus if this case study.

4.1 Design philosophy and inspiration

For several years Chess has researched and experimented with wireless sensor nodes using an epidemic communication model. This approach is inspired by biological systems such as ant colonies and social networking by humans. These systems demonstrate emergent behaviour where the whole is more than the sum of parts. This approach is reflected in the design of the MAC protocol with broadcast communication patterns, probabilistic and adaptive behaviour, redundancy, self-organisation, and a specification that is limited to local interactions.

4.2 Context

The gMAC as defined in this document is one of three protocol layers that are currently distinguished in the Chess Wireless Sensor Network. Atop the gMAC layer sits the Gossip layer

(see [Van Steen et al, Vrije Universiteit Amsterdam, Tech. Rep. IR-CS-012.05, 2005]) that is responsible for insertion of new messages, forwarding of current messages and deletion of old messages. The application layer, atop the Gossip layer, has the business logic that interprets messages and, optionally, generates new messages. The characteristics of these other layers influence the design decisions made in the gMAC layer. For instance, the highly redundant nature of the Gossip layer allows us to tolerate occasional message loss on the MAC layer. A future release of this document may include a specification of the Gossip layer.

4.3 Constraints

Our design space is constrained by specific hardware limitations and environment properties. These include:

- Interference in the much used ISM frequency band (Microwave, Bluetooth, Wifi, etcetera).
- Clock inaccuracies.
- Absence of collision detection possibilities. This means if a node does not receive a message in a particular period it can either be because no message was sent by any of the neighbouring nodes (if any) or that a collision occurred due to overlapping transmissions by neighbours or external interference.
- The topology of the network is unknown, sensors are arranged in a three dimensional space, the density of the network varies spatiotemporally.
- The range from which a node can receive messages is not necessarily the same as the range in which a node's message can be received. Hence, the neighbour relation is not necessarily symmetric.

4.4 Design choices

We list the design choices that have been made in the design of the Chess gMAC protocol outside those that are directly determined by the hardware constraints listed in the previous section:

- In order to meet the strict energy constraints, we use a Time Division Multiple Access (TDMA) MAC protocol. If we have some common notion of time, we can limit the period nodes are active and switch to an energy saving mode for the remainder of the time.
- There is no topology establishment and no routing mechanism. Although these are not services typically offered by the MAC layer, the fact that these services are also absent form higher layers does influence the design of the gMAC.
- All communication is broadcast with no receiver designation and no MAC-level node identifiers.
- There are no delivery guarantees on the MAC level and there is no investment in 2-way communication with acknowledgement messages. We assume redundancy in higher layers offers sufficiently high probability of message delivery. If there is interference within (a sub region of) the communication range of a node, the message will not be delivered to any node in that sub region.

4.5 Research questions

Currently the greatest challenge in the design of the gMAC protocol is to find suitable mechanisms for

- TDMA scheduling and
- Clock (re)synchronisation

Refer to sections 4.7 and 4.8 respectively for a specification of these problems. To evaluate the suitability of solutions we identify the following quality metrics:

- Resource consumption
  - Energy
  - Computational complexity
  - Storage
  - Bandwidth
- Latency: average time required to deliver a message to (all) its destinations
- Scalability: up to tens of thousands of nodes
- Robustness: tolerance to node and communication failure
- Mobility: nodes do not necessarily have a fixed location
- Adaptability: changing communication patterns and network density

Problem statement

Find a suitable solution for the above two challenges satisfying the constraints of section 4.3 offering a good trade-off between the quality metrics as defined above. If there are compelling reasons to deviate from the design choices as described in section 4.4, the solution should provide sufficient evidence that such deviation does not adversely affect the defined quality metrics.

4.6 Definitions and Terminology

Our TDMA MAC model is characterised as follows. Note that all references to time are based on the local perception of time. Time is divided in fixed length *frames*, each frame is subdivided in *slots*. S denotes the number of slots within one frame. $A<S$ denotes the number of active slots, that is the slots in which the node is either in listening mode for incoming messages from neighbouring nodes, or is sending a message. $D<S$ denotes the number of dormant slots, i.e. slots in which the node is neither sending nor listening. The following relation holds: $S=A+D$. In a typical scenario $A << D$ for reasons of energy conservation.

The active slots are placed in one contiguous sequence which, in the current implementation, is placed at a fixed location at the beginning of the frame. Hence we have a sequence of slots $(s_0, s_1, \ldots, s_{A-1}, s_A, \ldots, s_{S-1})$ such that

$V_A = \{s_i | 0 \leq i < A\}$ is the set of active slots and
$V_D = \{s_i | A \leq i < S\}$ is the set of dormant slots

4.7 TDMA scheduling

The TDMA schedule problem can now be stated as follows:

Determine for each time frame one slot $s_t \in V_A$ and one slot $s_{sync} \in V_D$. Slot s_t will be used to send a message and slots $V_A \backslash s_t$ will be used to listen for incoming messages from neighbouring nodes. Slot s_{sync} will be used to send a synchronisation message for nodes that are out of clock

synchronisation (for example nodes that have just joined the network). Refer to section 4.8 for a description of the synchronisation mechanism.

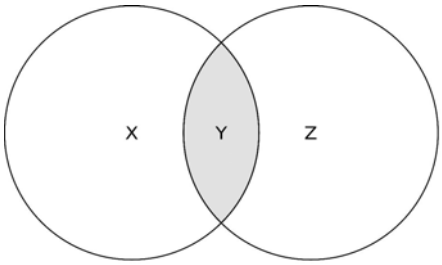
The slot allocation strategy should maximise the expected number of neighbours to which a message is successfully delivered. The slot allocation should also be fair. This means, for instance, that for each node the expected number of neighbours to which its message is delivered is "comparable" to that of nodes in a "comparable" situation (w.r.t. node density, amount of external interference etc.).

If two neighbouring nodes choose the same send slot, a collision will occur in the intersection of their ranges preventing message delivery of either node's message in that intersection. Ideally, no neighbouring pair would ever choose the same send slot. This has proven to be very hard to achieve, especially if node mobility is taken into account.

It is important to note that the number of neighbours may exceed the number of active slots A. Such a scenario might require nodes to be silent in some of their allocated send slots, as too many collisions hampers all communication.

Example

To illustrate the scenarios the TDMA slot allocation should take into account we give the "Hidden node problem" as an example. Consider three nodes in a linear arrangement such that the middle node Y is within the range of both other nodes. The outer nodes X and Z are outside each others range. This setup is visualised in the picture below.



If nodes X and Z were to choose the same send slot, their messages would collide in the intersection of their ranges. Hence node Y will receive neither message and, in the absence of collision detection, cannot distinguish this from the situation where no message was sent. Neither node X nor Z can autonomously detect this conflicting slot allocation even if they would occasionally listen instead of send in their allocated send slot. The traditional solution to this problem, as described in [Bharghavan et al, *"MACAW: a media access protocol for wireless LANs"*, SIGCOMM'94, ACM, p212-225], uses a 2-way communication in which the intended receiver (Y), on request of the sender (X), sends a message announcing to all nodes (e.g. Z) within the range of the receiver the forthcoming transmission. This solution is inappropriate for our scenario as all communication is broadcast and we want to avoid the negotiation overhead. Instead we use a piggybacking solution where each message includes the sender's perspective on the current slot allocation. In the previous example node Y would report the slot in which both X and Z send as a vacant slot from which X and Z can conclude that a slot allocation conflict has occurred. See below for the specification of the message format including the description of the slot allocation.

4.7.1 Message format

MAC messages are defined as the sequence *(ID, n, T, P, MDC(ID,n,T,P))* where

- *ID* is a domain identification string which allows to disregard messages from a different administrative domain.
- *n* denotes the slot sequence number (from the sender's perspective) in which the message was sent.
- *T* is the sender's perspective on the current slot allocation. *T* is defined as the sequence (*b0,b1,...,bA-1*) such that $bi \in \{true, false\}$ for $0 \leq i < A$. *bi=true* implies that from the sender's perspective slot *i* is occupied, either because it is the sender's send slot or because the sender has received a message in slot *i*. *bi=false* implies that form the sender's perspective slot *i* is vacant. Recall that in case of a collision in slot *i*, the sender also considers it vacant.
- *P* is the fixed length MAC payload message.
- *MDC* is a Modification Detection Code parameterised by the preceding elements.

In our current implementation the byte sizes of these elements are as follows:

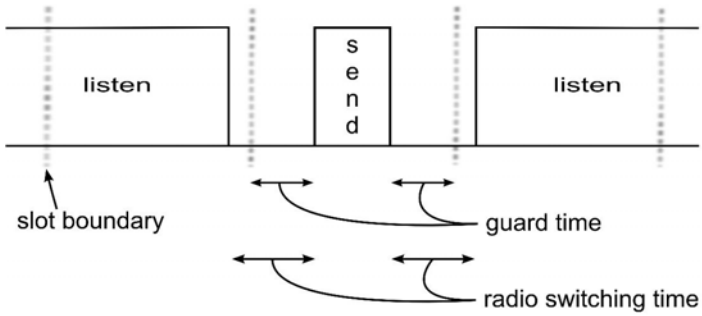
ID	3
n	2
T	1
P	29
MDC	2

4.7.2 Implementation issues

Because the nodes have imperfect local clocks, over time the (perspective on the) frame boundaries will drift apart. Even with frequent resynchronisation, some margin has to be introduced such that the send interval does not fall outside the receiver's listen interval. We refer to this margin as the *guard time*.

For a contiguous sequence of listening slots the radio remains in listening mode such that out of sync transmissions from other nodes that cross slot boundaries are also received.

Switching the radio from listening to transmitting mode requires a certain amount of time during which we can neither receive nor send. Currently, this switching time exceeds the guard time, leading to the timeline illustrated below.



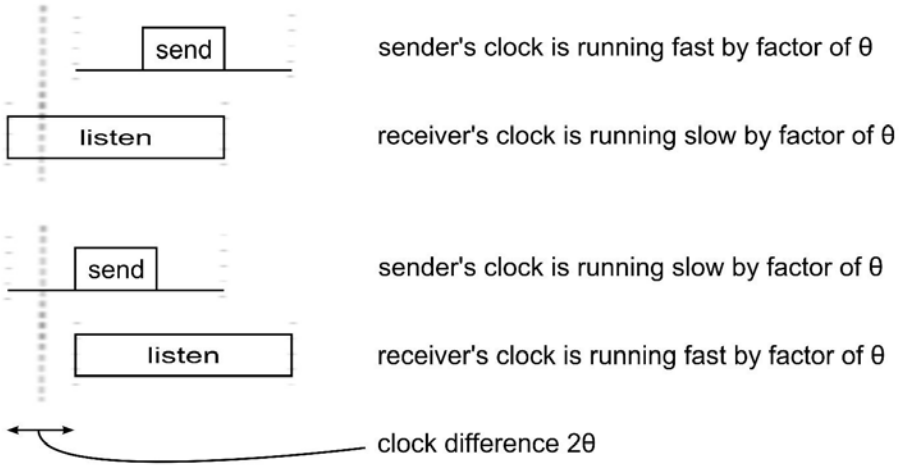
4.8 Clock synchronisation

As follows from the description above, the slotted MAC scheme requires nodes to be loosely synchronised. It suffices for nodes to have a common notion of the frame boundaries (and by extension the slot boundaries). Synchronisation with an external reference clock (e.g. UTC) is not required. The smaller the clock difference between sender and receivers, the smaller the slot guard times can be. Smaller guard times allow for smaller slot sizes (for fixed message sizes) which reduces the time a node has to be in the energy consuming active state. Alternatively, with the same energy consumption the number of active slots can be increased which improves network throughput.

The maximum clock drift rate θ allows us to derive a lower bound on the required guard time t_g. If we assume clocks were perfectly synchronised during the active period of the previous frame, the maximum clock difference is given by

$$T_f \cdot 2\theta$$

where T_f is the frame length (i.e. the time elapsed since the moment of perfect synchronisation) and the factor 2 reflects the worst case where clocks have drifted in the opposite direction. This situation is depicted in the figure below. The first receiver's clock is running slow by a factor of θ. If we assume the sender's clock is running fast by the same factor, the absolute time difference is given by the expression above. This justifies the guard time at the end of the sender's slot. Similarly, for a fast receiver in combination with a slow sender, the guard time at the beginning of the (sender's) slot is required for successful message receipt.



We have

$$t_g \geq T_f \cdot 2\theta$$

as the theoretical lower bound for the guard time. This implies that at least $2t_g$ is required per slot to compensate for clock drift (guard time at both beginning and end of the slot). During this time the receiver is in the power consuming listening mode but not actually receiving messages.

We remark that also senders should be loosely synchronised such that no collisions occur due to clock differences (rather than TDMA schedule conflicts). However, clock synchronisation

requirements between senders are lower by a factor of 2. Consider two perfectly synchronised senders that have adjacent send slots. Then there is a time difference of $2t_g$ between the end of the one transmission and the beginning of the other. Hence a clock difference of $2t_g$ or $4\theta T_f$ can be tolerated. Consequently, any satisfactory sender-receiver synchronisation mechanism will certainly satisfy sender-sender synchronisation requirements.

In practice we cannot assume to have perfect initial synchronisation. Moreover, the clock synchronisation mechanism should take more than one sender-receiver pair into account. As we typically have more than one receiver per sender it is challenging to establish consensus on the frame boundary. Based on these observations we expect the guard time required for efficient communication to be considerably higher than the derived theoretical lower bound.

Note that it is acceptable that the common agreement on the frame boundary shifts gradually from one side of the network to the other as long as nodes within a communication range have an appropriately low clock difference.

The clock synchronisation mechanism can use the following information to determine local clock adjustments.

- Each message contains the slot number *n* in which it was transmitted (in the perception of the sender)
- The time *td* at which a message is delivered is determined by the sender's perception of the slot start time *t0*, the guard time *tg*, the message transmission time *tm* and the propagation delay *tp*. Assuming (estimates for) *tg*, *tm* and *tp* are known, the sender's *t0* can be determined.

For initial synchronisation a node can use the messages sent by each node in the *ssync* slot. This allows each node to listen only in the active period, even if this period does not overlap with the active period of other nodes. In our current implementation slot *ssync* is chosen randomly in the TDMA schedule.

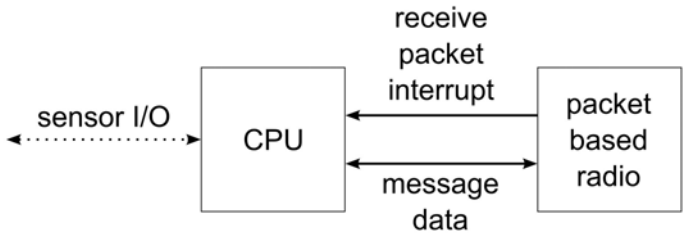
The clock synchronisation problem can now be stated as follows:
Based on the information that is available from the existing messages as described above, determine a clock synchronisation mechanism offering

- Rapid distributed initial synchronisation
- Continued resynchronisation taking node mobility into account (including joins of previously disconnected networks)
- Fast convergence
- Robustness against benign or malicious desynchronising input (faulty node, active MAC-level adversary)
- Good trade-off between length of guard time and overall network performance

4.9 Implementation issues

In our current implementation where $T_f = 1$ s, $\theta = 20$ ppm, $S = 1129$ and the oscillator frequency is 32768 Hz we have at a minimum 4 cycles of the total of 29 cycles/slot as guard time. We actually use 18 cycles as guard time, the remaining 11 cycles are required to transmit the 34 byte message.

Currently, the radio subsystem issues an interrupt to the CPU upon successful receipt of a message. The oscillator is used to determine the time of message delivery. Hence t_d is only accurate w.r.t. the 29 cycles per slot.



The table below gives some performance characteristics of our current implementation:

Radio current transmission	11.3 mA
Radio current receiving	12.3 mA
Radio current idle	0.9 μA
CPU current operational	0.5 mA
CPU current idle	0.01 mA

5 TERMA: Herschel/Planck software

This case study considers the ACC ASW software, a system for satellite attitude and orbit control used within the Herschel and Planck satellite systems.

Details of this case study are covered by a distinct non-disclosure agreement

The information below is public.

Based on sensor data, the satellite's attitude is determined. Using this information, the deviation from the reference attitude is determined. The result is given to a controller that calculates the torque necessary to achieve the desired attitude. This torque is then realized through commands to actuators. For orbit control, ground commands an attitude and a required change of velocity, and the system fires the orbit control thrusters accordingly.

The health of sensors, actuators and other important control system units are monitored, and actions taken when errors are detected. These actions comprise (autonomous) re-configuration and error reporting.

All attitude control related events are reported to ground along with system housekeeping and diagnostic telemetry, to reflect the status of the attitude control system.

Finally, uploaded attitude control related commands, for example requesting control re-configuration or certain sensor/actuator activity, are processed and executed, and verification reports on their successful or unsuccessful execution are downloaded to ground.

5.1 ACC ASW Architecture

The architecture of the Herschel/Planck ACC ASW reflects the aim of maximizing the commonality between the Herschel and the Planck software. The architecture is primarily Herschel-based, since it generally has more sensors and actuators than Planck does.

The architecture has been described using UML, as supported by the Rational Rose tool, which was a requirement from our customer. Because the project started before the UML RT profile was mature, we introduced our own stereotypes to capture real-time properties of classes/objects. These stereotypes were heavily inspired by the Hard Real-Time HOOD method [HRT-HOOD], which Terma Space has used on previous projects, since it allows a systematic analysis of the real-time properties of a software system.

The ACC ASW architecture consists of a number of inter-related subsystems. The subsystems and their main interrelations are shown in the figure below. Dependencies directed at Failure Detection, except for the one from Data Acquisition that designates HK collection, have been left out to avoid cluttering the diagram.

The diagram contains sensor subsystems on the left, subsystems for control and supervision in the middle and actuator subsystems on the right. External interfaces (to ACC BSW and RTEMS) are placed at the bottom and at the top of the diagram.

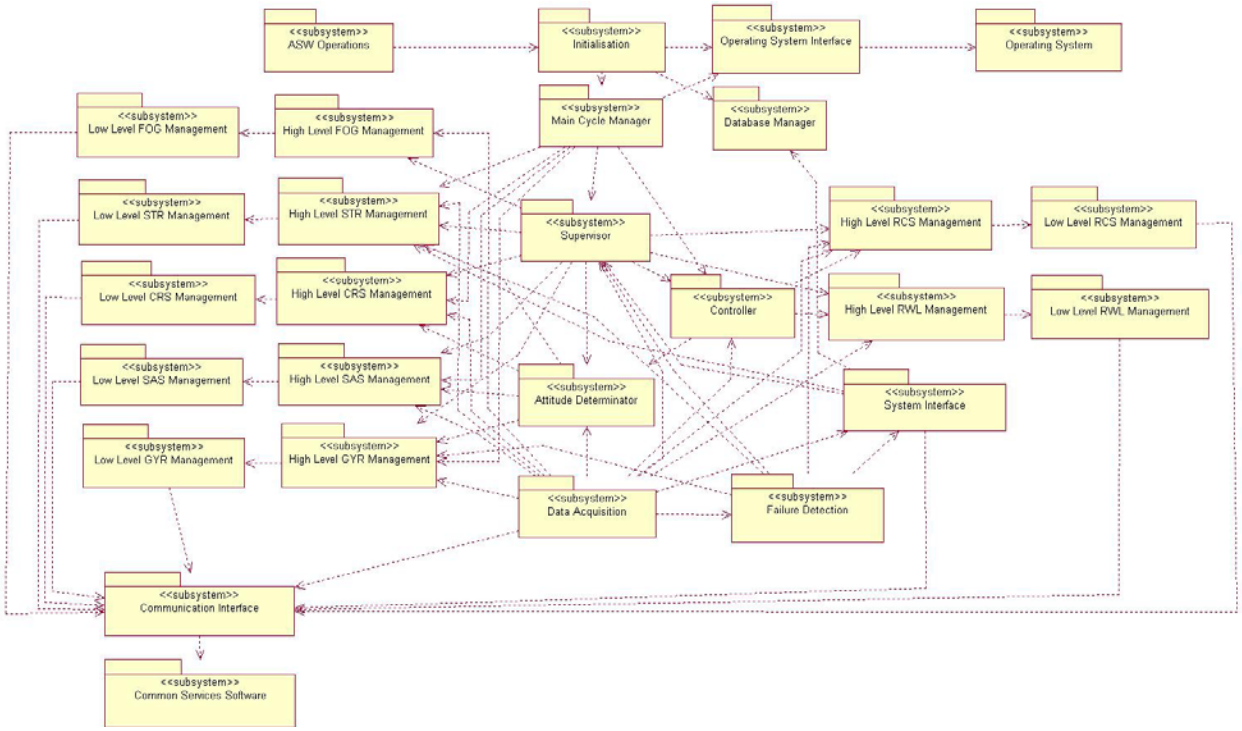


Figure 5-1: Top-Level ACC ASW Architecture

The subsystems are:

- Initialisation, which is responsible for the ACC ASW initialisation and starting of the main cycle.
- ASW Operations, which defines the operations that are to be called by ACC BSW for activating ASW and requesting a context saving. This subsystem constitutes the complete external interface provided by ACC ASW.

- Main Cycle Manager, which controls the cyclic activities.
- Database Management, which handles access to and updates of the onboard database and performs periodic CRC check of the contents.
- High Level Coarse Rate Sensors Management, which handles high-level control, data conversions and local fault detection for the coarse rate sensors (CRS).
- Low Level Coarse Rate Sensors Management, which handles low-level data retrieval for the coarse rate sensors (CRS).
- High Level Gyroscope Management, which handles high-level control, data conversions and local fault detection for the gyroscopes (GYR), Herschel only.
- Low Level Gyroscope Management, which handles low-level commanding and data retrieval for the gyroscopes (GYR), Herschel only.
- High Level Reaction Control System Management, which handles high-level control, data conversions and local fault detection for the reaction control system (RCS).
- Low Level Reaction Control System Management, which handles low-level commanding and data retrieval for the reaction control system (RCS).
- High Level Reaction Wheels Management, which handles high-level control, data conversions and local fault detection for the reaction wheels (RWL), Herschel only.
- Low Level Reaction Wheels Management, which handles low-level commanding and data retrieval for the reaction wheels (RWL), Herschel only.
- High Level Star Tracker Management, which handles high-level control, data conversions and local fault detection for the star trackers (STR).
- Low Level Star Tracker Management, which handles low-level commanding and data retrieval for the star trackers (STR).
- High Level Sun Sensor Management, which handles high-level control, data conversions and local fault detection for the sun sensors (SAS).
- Low Level Sun Sensor Management, which handles low-level data retrieval for the sun sensors (SAS).
- High Level FOG Management, which handles high-level control, data conversions and local fault detection for the FOG, Planck only.
- Low Level FOG Management, which handles low-level commanding and data retrieval for the FOG, Planck only.
- Attitude Determinator, which implements mission- and mode-dependent attitude determination.
- System Interface, which receives and checks incoming telecommands and executes most of them, and provides support for event and verification report generation.
- Supervisor, which handles reconfiguration, telecommanding, mode transitions, and set-point generation.
- Controller, which implements mission- and mode-dependent control algorithms to determine required actuation.
- Data Acquisition, which initiates data collection for housekeeping and diagnostics reporting.
- Failure Detector, which provides failure detection at system level. Most of the “Recovery” part of FDIR is handled by the Commander component of the Supervisor.

- Communication Interface, which provides an interface to the ACC BSW services.
- Operating System Interface, which provides an interface to the RTEMS routines used by ACC ASW.
- Common Services Software, which describes the ACC BSW services provided to ACC ASW, i.e. it is an external interface that is not further described.
- Operating System, which describes the RTEMS services used by ACC ASW, i.e. it is an external interface that is not further described.

A number of system level design paradigms have been driving the design outlined above:

- *System level mode transitions* - and resulting reconfigurations - are centralised (in Supervisor) to maintain consistency and base autonomous transitions on a system level perspective.
- *Failure detection* is distributed across the system thus accommodated as close to the originating source as possible.
- *Failure isolation and recovery* takes place at system level, thus being centralised (in Supervisor).
- *All external interfaces* are encapsulated in dedicated interface components (Communication Interface and Operating System Interface) to mitigate consequences of external failures as well as modifications of interfaces.
- General *control engineering* is decoupled from vendor specific actuator and sensor characteristics, thus operating on calibrated engineering values (achieved by having "high-level" and "low-level" management subsystems for all sensors and actuators).

5.2 ACC ASW Process Structure

The ACC software consists of a number of tasks, some of which are part of ACC BSW and some of which are part of ACC ASW. This description only deals with the latter. However, when performing a schedulability analysis, the whole task population of course has to be taken into account.

All tasks within ACC ASW are part of the Main Cycle Manager subsystem, i.e. all other subsystems contain purely sequential code that is invoked, directly or indirectly, by one of the ACC ASW tasks.

The Main Cycle Manager subsystem contains five tasks:

- Main Cycle, which is a cyclic task, controlling and monitoring the more detailed operations of the main control cycle.
- Primary Functions, which is a sporadic task performing sensor data acquisition, attitude determination, guidance and control.
- RCS Control Functions, which is a sporadic task that commands the RCS.
- Secondary Functions, which is a sporadic task, performing the remaining control functions, as TC handling, diagnostics, TM processing and configuration management.
- Recovery, which is a sporadic task that handles any reconfiguration that takes place as part of FDIR.

The Main Cycle Manager does not perform the operations itself, it merely orchestrates the cyclic operation of the ACC ASW, and monitors that the various tasks keep their deadlines.

Each of the above tasks has been described using UML activity diagrams. A couple of those are included below, to give the reader an impression of the way the ACC ASW has been described.

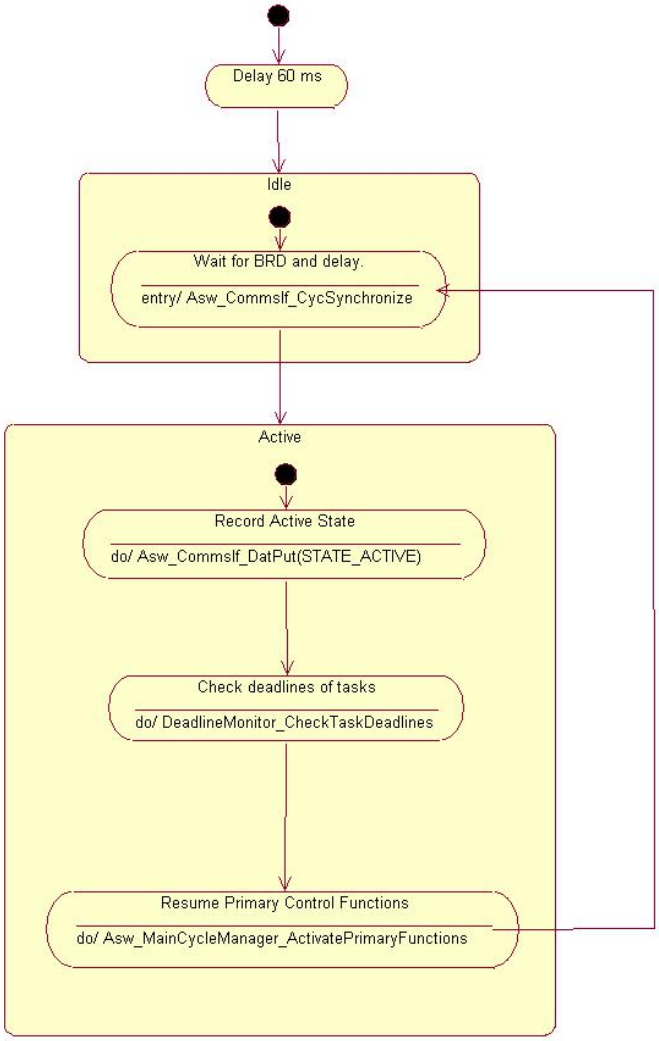


Figure 5-2: Main Cycle Activity Diagram

The activity diagram shown in Figure 5-2 states that before the cyclic processing starts, a 60 ms delay is performed (in order to ensure that the ACC BSW is in a well-defined state). After that the task is either Idle or Active, where it in the Idle state is waiting to be periodically released by ACC BSW with a period of 250 ms. In the Running state it performs a few status recordings and checks, and then activates the Primary Control Functions task, which is described below.

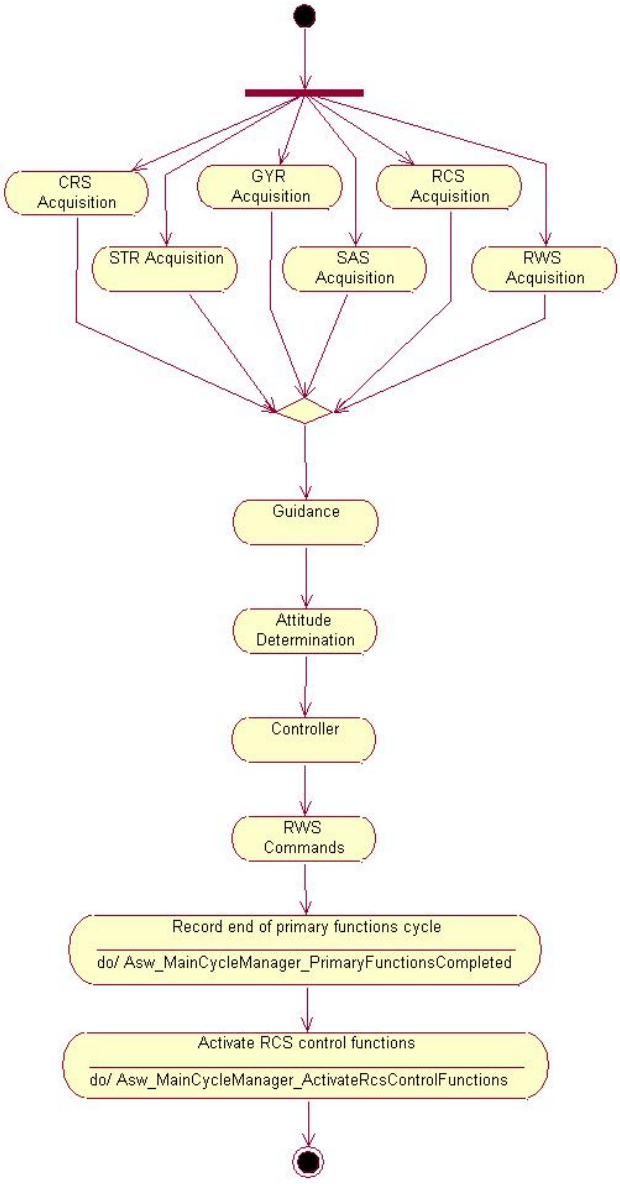


Figure 5-3: Primary Control Functions Activity Diagram

The activity diagram shown in figure Figure 5-3 states that first data is acquired (in some order) from all sensors and actuators. Then Guidance is performed (Herschel only), after which the current attitude is determined, the appropriate controller determines the required actuation to be performed, the Reaction Wheel System (RWS) is commanded (Herschel only), and finally the Reaction Control System (RCS) task is activated.

5.3 FDIR Approach

Failure Detection, Isolation and Recovery (FDIR) in the HP ACC ASW is both centralised and decentralised. The functionality necessary for an efficient FDIR mechanism is performed by dedicated components as well as all other components, depending on the nature of the activity. The different activities are performed as described below.

- Unit checks are all performed in each High Level Sensor Manager. Some checks, the Str Gyr Cross Check and the Gyr Diagnostics Check are performed by the Failure Detection

component. If data replacement needs to be done, for instance because of failing unit checks, this is done by each respective High Level Manager.

- Enabling/disabling of unit checks is done by a Unit Checks Manager, which is a part of the Supervisor. The manager reads the safeguard memory and reconfiguration module to find out which checks are enabled and then commands each High Level Sensor Manager accordingly.
- Unit check result processing is done in the Failure Detection component. The processing results in each unit being declared healthy/unhealthy.
- Recovery operations, where an unhealthy unit is replaced by a healthy one, are performed by the Unit Commanding module. The Unit Commanding module is a part of the Commander, which is part of the Supervisor.
- Sporadic errors are detected in the affected module, which then calls Failure Detection to report the error. If any data replacement needs to be done, for instance inside an algorithm, so that the algorithm can finish, then this is performed by the component itself.

5.4 Research Questions

Since the ACC ASW is an embedded real-time system, a number of quantitative requirements are applicable. Most of these requirements are related to time, either in the form of deadlines of activities or in the form of maximal allowed CPU load.

Examples of such requirements imposed by our customer are:

- Herschel ASW shall not exceed 45% (measured) occupancy of CPU load and Planck ASW not exceed 50%. Note: Included is the execution of BSW SVCs.
- The ASW shall read unit data via the BSW, starting 20ms after the broadcast. Note: This requires that the BSW has finished reading of units before 20ms after the BC.
- The ASW shall complete all calculations needed to determine RWL commands and provide them to the BSW before 70 ms after the broadcast.
- The ASW shall complete all tasks allocated to the cycle within 250ms after the broadcast.
- The execution time between start of ASW initialisation and entry of Survival Mode shall be less than 0.25 s. Note: this time is critical for recovery of serious failures that require that RCS control be established as soon as possible. Total start-up time is budgeted as 4.5 s.
- The execution time between start of ASW initialisation and entry of Nominal Mode shall be less than 10 s.

In addition, there are requirements on memory usage, as for example:

- The ASW shall not exceed 350 Kbytes of EEPROM at each PDR and 500 Kbytes at each QR.
- The ASW shall not exceed 1.2 Mbytes of RAM at each PDR and 1.6 Mbytes at each QR.