MiXiM - Physical Layer

Karl Wessel: wessel@tkn.tu-berlin.de
Michael Swigulski: swigulski@tkn.tu-berlin.de

20. November 2007

1 requirement specification

1.1 Overview

- provide status information to MAC
- switch radiostate (RX, TX, SLEEP)
- send packets to air/channel
- receive packets / listen for packets
- provide hooks for statistical information
- configurable settings

1.2 provide status information to MAC

In addition to received packets the physical layer has to provide some other information to the MAC layer. Some of this information has to be provided passively on demand (e.g. current channelstate) and some should be delivered actively to the MAC layer on certain events (e.g. transmission of a packet complete).
Information which has to be provided to MAC on demand:

- channelstate: busy/idle (boolean) or RSSI
- current radiostate (RX, TX, SLEEP)

Information which has to be provided to MAC the moment it occurs:

- transmission over (send)

1.3 switch radiostate

The physical layer has to be able to switch between the following things:

- current radiostate (RX, TX, SLEEP)

Switching from one state to another may take some time. Whereas the switching time may depends to whitch state we are switching.

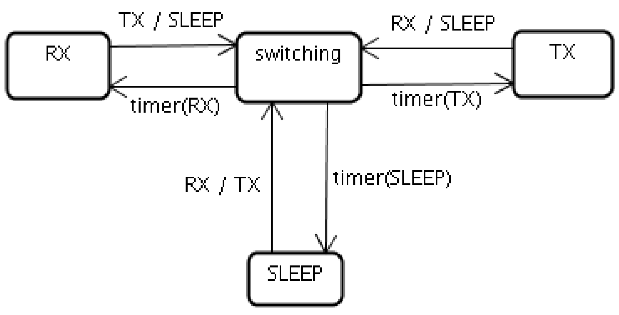

Figure 1: State machine for radiostate

1.4 send packets

The physical layer has to be able to send packets from the MAC layer to the channel. We also want to support the possibility to control the sending process after it has been started.
Before we can send packets the following things have to be assured:

- the radio has to be in TX state
- we are not already sending

The above items should be controlled by the MAC layer so the physical layer would only throw an error if they are not set.
The following information has to be attached by the MAC layer to the packet:

- channel dimensions (frequency/space)
- bitrate (for all dimensions) over time
  Note: Most cases will be covered by header and payload bitrate.
- TX power for all dimensions over time
- size of packet

The sending process itself is made up of the following steps:

1. MAC layer gives packet and control info to physical layer
2. check requirements for sending, throw error if they are not fulfilled
3. add information needed by the receiving physical layer to packet (see below)
4. add signal to packet
5. packet is sent to channel by physical layer
6. schedule transmission over message for MAC layer

The following information is needed by the receiving physical layer:

- signal
  - TX power (for all dimensions) over time
  - the bitrate (for all dimensions) over time
  - the channel(dimensions)
  - the duration the signal would need to be transmitted
  - the duration of the preamble
- position, move direction and speed of the sending host
- size of packet

1.5 receive packets

Because the packets arrive immediately at every receiving node we have to simulate the receiving process:

1. simulate propagation delay (if needed)
2. simulate preamble duration
3. simulate payload duration

We also have to simulate the attenuation of the signal strength. This should be done by filtering the signal with the *analogue models* directly after a message arrives (since we already have it then). The AnalogueModels will add an attenuation matrix to the Signal.

signal or *noise*. The decision is made by the *Decider*. Therefore the preamble has to be filtered previously by the analogue models.
The packet has to be classified as *signal* or *noise*. The decision is made by the *Decider*.

If the transmission of a *signal* is over we have to decide if it was received correctly. This is also done by the *Decider* by evaluating the *signal to noise ratio* short *SNR*. If the signal was received correctly we pass it to the MAC layer. It is also possible to pass the signal to the MAC layer anyway, marked as biterror/collision.

1.5.1 the analogue model

The *analogue model* simulates the attenuation of the signal strength by filtering the receiving power function.
There should be models to simulate the following things:

- pathloss
- shadowing
- fading

Further we set the following requirements to the *analogue models*:

- physical layer should be able to apply multiple *analogue models* to a signal
- you should be able to set the *analogue models* independent from physical layer
- you should be able to add your own *analogue models*

1.5.2 the Decider

As mentioned already above the *Decider* has to decide the following things:

- classify packet as *signal* or *noise*
- decide if a packet was received correctly on the basis of the signal and interfering noise

We set the following requirements to the *Decider*:

- you should be able to set the *Decider* independently from physical layer
- you should be able to add your own *Decider*
- the *Decider* should be able to return bitwise correctness of the *signal* (on demand)

1.6 statistical information

You should be able to get the following statistical information (the physical layer should not evaluate it but has to provide access to the according information):

- packet count
- received signal strength
- signal to noise ratio
- bit error ratio
- collisions

1.7 parameters

The following parameters of the physical layer should be freely configurable:

- simulate propagation delay? (boolean)
- which analogue models should be used
- the parameters for the analogue models
- which Decider should be used
- the parameters for the decider
- thermal noise
- sensitivity
- maximum TX power
- switching times between states (RX, TX, SLEEP)

2 modelling

Note: We denote a Layer in a general meaning by 'physical layer' or 'MAC layer' and our concrete C++ classes by *BasePhyLayer* or *BaseMacLayer*.

2.1 overview

Here we present the design- and interface details of the OMNeT-module *BasePhyLayer* to meet the requirement specification. That includes:

1. internal class diagram of *BasePhyLayer* and relation to *BaseMacLayer*
2. interface description for all involved C++ classes
3. flow charts for reception of MacPacket from upper layer and AirFrame from the channel
4. some detailed flow charts for important sub processes

2.2 classgraph

We start with the classgraph for the OMNeT-module *BasePhyLayer* that shows its C++ classes, relations to other OMNeT-modules (especially *BaseMacLayer*) and the OMNeT-messages sent between them.
The *BasePhyLayer* holds a list(req. 1.5.1) of AnalogueModels and a pointer to a Decider. Thus the AnalogueModel and the Decider are submodules of *BasePhyLayer*. This way one is able to change(req. 1.5.1)(req. 1.5.2) and replace(req. 1.5.1)(req. 1.5.2) them independently from the *BasePhyLayer*.

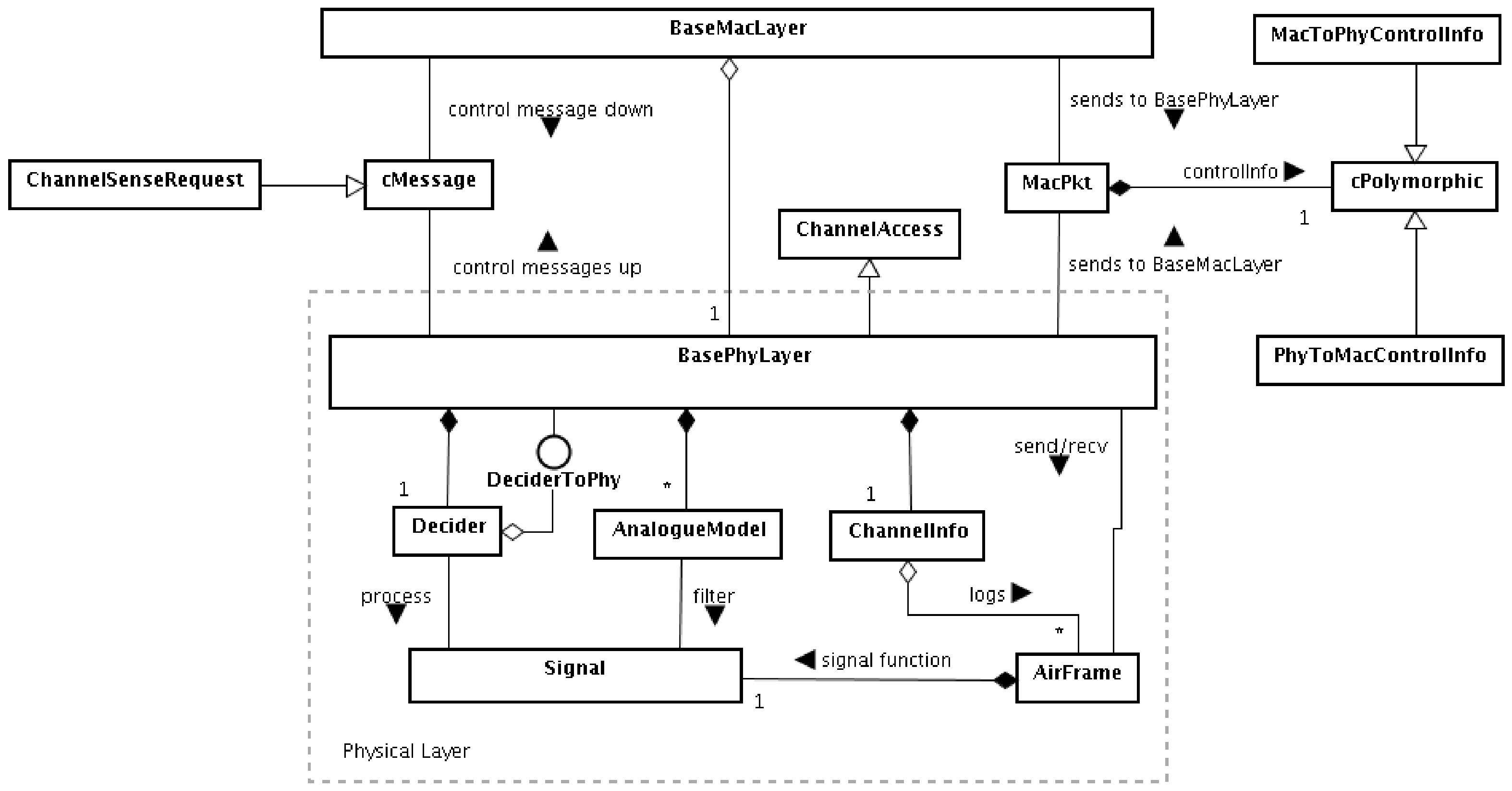

Figure 2: class graph

2.3 The *BasePhyLayer* interface

In this section we focus on how one is able to communicate with the *BasePhyLayer*, i.e. especially the *BaseMacLayer* which is connected to the *BasePhyLayer* in three ways:

1. OMNeT-channel for data messages
2. OMNeT-channel for control messages
3. a reference to *BasePhyLayer*.

The data channel is used to send and receive$^{(\text{req. } 1.4)}$ MacPkts to and from the *BasePhyLayer*. An appropriate ControlInfo is attached to the packet by the sending layer. *see also Figure 9*

The control channel is used by the *BasePhyLayer* to inform the *BaseMacLayer* about certain events$^{(\text{req. } 1.2)}$, like the TX_OVER$^{(\text{req. } 1.2)}$ message which indicates the end of a sending transmission.
A ChannelSenseRequest can be sent over control channel from *BaseMacLayer* to *BasePhyLayer*. It is handed to the Decider (several times) that attaches a ChannelState and finally sent back to *BaseMacLayer*.

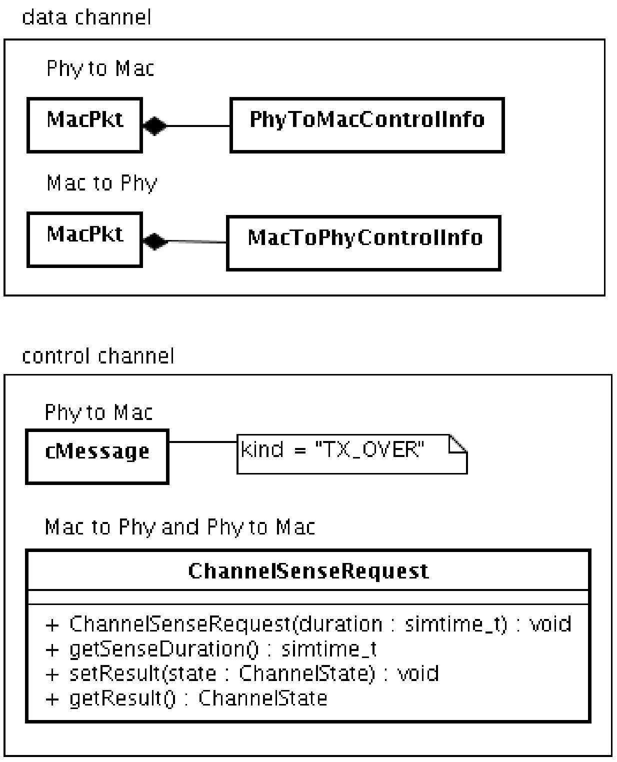
data channel
Phy to Mac
MacPkt
PhyToMacControlInfo
Mac to Phy
MacPkt
MacToPhyControlInfo
control channel
Phy to Mac
cMessage
kind = "TX_OVER"
Mac to Phy and Phy to Mac
ChannelSenseRequest
+ ChannelSenseRequest(duration : simtime_t) : void
+ getSenseDuration() : simtime_t
+ setResult(state : ChannelState) : void
+ getResult() : ChannelState

Figure 3: messages sent between *BasePhyLayer* an *BaseMacLayer*

The reference provides a passive way$^{(\text{req. }1.2)}$ for the *BaseMacLayer* to get information about the current channel state$^{(\text{req. }1.2)}$ (that is an alternative [immediate answer] to sending a ChannelSenseRequest) and to get$^{(\text{req. }1.2)}$ and set$^{(\text{req. }1.3)}$ the current radiostate (RX, TX, SLEEP). Switching times$^{(\text{req. }1.3)}$ from one radio state to another are controlled internally by a state machine. *see also Figure 1*

The createSignal method returns *BaseMacLayer* a reference to a Signal at a minimum of necessary parameters, it only has to pass txPower, headerBitrate and payloadBitrate. This is the key method for *BaseMacLayer* in order to create MacToPhyControlInfo properly.

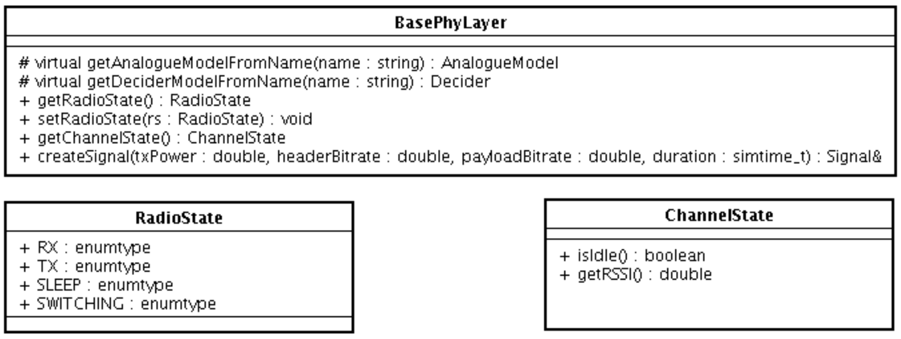
BasePhyLayer
# virtual getAnalogueModelFromName(name : string) : AnalogueModel
# virtual getDeciderModelFromName(name : string) : Decider
+ getRadioState() : RadioState
+ setRadioState(rs : RadioState) : void
+ getChannelState() : ChannelState
+ createSignal(txPower : double, headerBitrate : double, payloadBitrate : double, duration : simtime_t) : Signal&
RadioState
+ RX : enumtype
+ TX : enumtype
+ SLEEP : enumtype
+ SWITCHING : enumtype
ChannelState
+ isIdle() : boolean
+ getRSSI() : double

Figure 4: BasePhyLayer interface

2.4 AnalogueModel

The AnalogueModel is at least able to filter the *TX-Power over time*-Signal.

Information how it works on a more complex multi-dimensional Signal are explained in section 2.11.

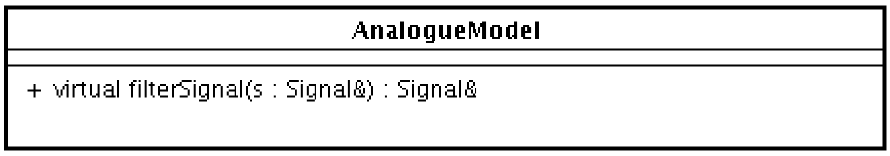

Figure 5: analogue model interface

2.5 ChannelInfo

ChannelInfo keeps track of all AirFrames on the channel. It does not differentiate between *signal* and *noise*. *BasePhyLayer* is able to add and remove references to certain AirFrames to and from ChannelInfo.

ChannelInfo can return a vector of Signals (references) that intersect with a given time interval.

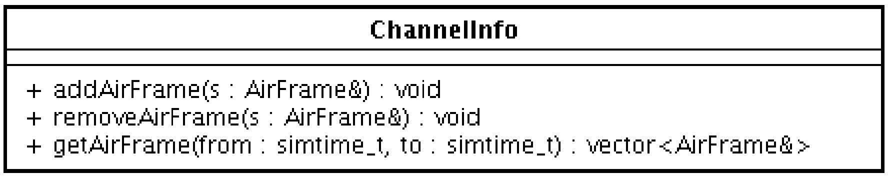
ChannelInfo
+ addAirFrame(s : AirFrame&) : void
+ removeAirFrame(s : AirFrame&) : void
+ getAirFrame(from : simtime_t, to : simtime_t) : vector<AirFrame&>

Figure 6: channel details

2.6 Decider and DeciderToPhy interface

The main task of the Decider is to decide which packets should be handed up to the MAC layer. To achieve this the *BasePhyLayer* hands every receiving AirFrame several times to the “processSignal()“- function of the Decider.
The most common case how a Decider would process AirFrames would be the following:

1. If the Decider gets an AirFrame for the first time, it determines the time point it can decide if the packet is noise or not and returns this time point to the *BasePhyLayer*. The time point could be the preamble length, for example.

2. The next time the Decider gets the packet it would internally decide if the packet has to be considered noise or not. If the decision is noise the packet isn't interesting anymore for the Decider. If the packet is classified as a signal the Decider would return the end of the signal to the *BasePhyLayer*.

3. When the receiving of the AirFrame is over and the AirFrame wasn't classified as noise, the Decider would decide if the packet was received correctly. If the result is positive the Decider has to tell the *BasePhyLayer* to send the AirFrame together with the DeciderResult to the *BaseMacLayer*.

A second task of the decider is to decide if the channel is busy or idle at a specific point in time or during a given interval.

Since the Decider is responsible to decide if an AirFrame should be decapsulated and handed up to the MAC layer the Decider needs an interface to the *BasePhyLayer*.

Over the interface the Decider can do the following things:

- get the current simulation time

- get the list of AirFrames which intersect which a specific interval (to calculate SNR)

- tell the *BasePhyLayer* to hand an AirFrame up to the MAC layer

- tell the *BasePhyLayer* to send a control message to the MAC layer

Due to the last point the Decider is able to answer a ChannelSenseRequest of the MAC layer.

A Decider that gives a more detailed DeciderResult (e.g. bitwise errors$^{(\text{req. } 1.5.2)}$) must be subclassed and implemented by the user.

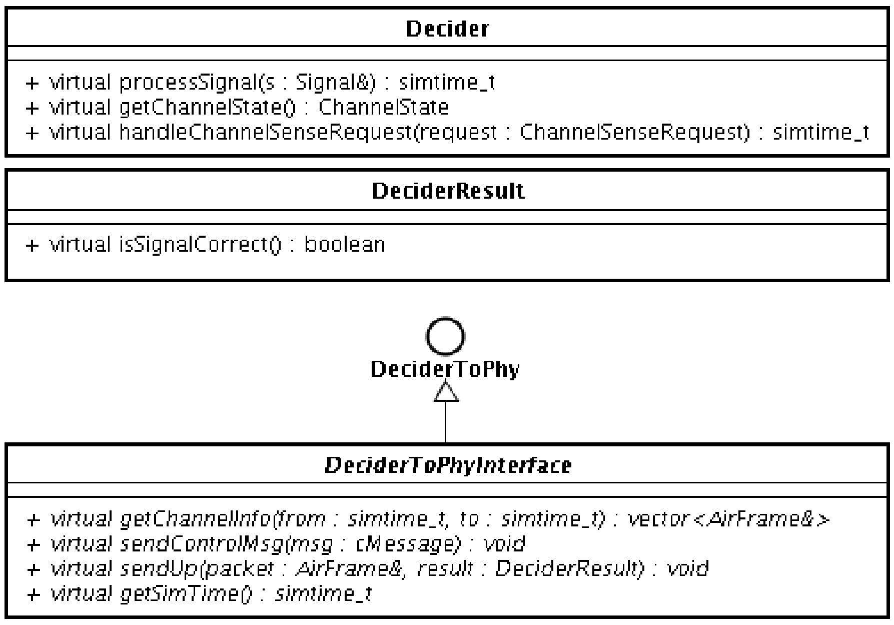
Decider
+ virtual processSignal(s : Signal&) : simtime_t
+ virtual getChannelState() : ChannelState
+ virtual handleChannelSenseRequest(request : ChannelSenseRequest) : simtime_t
DeciderResult
+ virtual isSignalCorrect() : boolean
DeciderToPhy
DeciderToPhyInterface
+ virtual getChannelInfo(from : simtime_t, to : simtime_t) : vector<AirFrame&>
+ virtual sendControlMsg(msg : cMessage) : void
+ virtual sendUp(packet : AirFrame&, result : DeciderResult) : void
+ virtual getSimTime() : simtime_t

Figure 7: Decider interface

2.7 The one-dimensional Signal and AirFrame

AirFrame and Signal both hold information about the packet to send. While the AirFrame is responsible for the OMNeT related data which is necessary to send OMNeT messages, the Signal holds the data which is necessary for the simulation of the transmission process.

Here we focus on the basic, one-dimensional (time) Signal that stores entries for TX Power and attenuation over time[1]. There are fix entries for header and payload bitrate. These assumptions are considered to cover most cases. It is possible to obtain a time iterator for this Signal to have access to entries at specific time points.
Further Signal stores the Move (move pattern of the sending host), the packets header length, the start time and length of the Signal.

Note: The multi-dimensional Signal is an instance of the same class but has more functionality. It is described in detail in a separate section.

To be able to control the sending process$^{(\text{req. } 1.4)}$ of an AirFrame every AirFrame has a unique id and a specific type which specifies if this is a normal AirFrame or a control-AirFrame. Further it holds an instance of Signal which represents the physical signal.

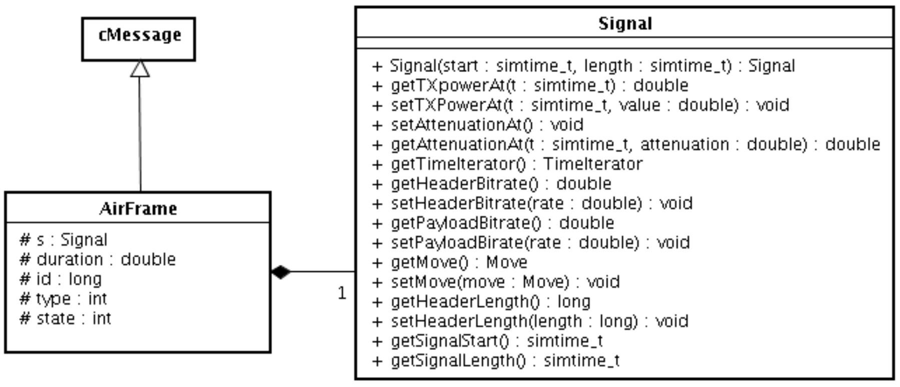
cMessage
AirFrame
# s : Signal
# duration : double
# id : long
# type : int
# state : int
1
Signal
+ Signal(start : simtime_t, length : simtime_t) : Signal
+ getTXpowerAt(t : simtime_t) : double
+ setTXPowerAt(t : simtime_t, value : double) : void
+ setAttenuationAt() : void
+ getAttenuationAt(t : simtime_t, attenuation : double) : double
+ getTimeIterator() : TimeIterator
+ getHeaderBitrate() : double
+ setHeaderBitrate(rate : double) : void
+ getPayloadBitrate() : double
+ setPayloadBirate(rate : double) : void
+ getMove() : Move
+ setMove(move : Move) : void
+ getHeaderLength() : long
+ setHeaderLength(length : long) : void
+ getSignalStart() : simtime_t
+ getSignalLength() : simtime_t

Figure 8: member arrangement in AirFrame and Signal

[1]The time stamps are values relative to starting time point

2.8 receiving a MacPkt

On reception of a MacPkt from the MAC layer, *BasePhyLayer* checks if:

1. the radio is in TX state$^{(\text{req. } 1.4)}$,
2. it is not already sending a packet$^{(\text{req. } 1.4)}$

If one of these conditions is not fulfilled it will throw an error.

The MacToPhyControlInfo object attached to the MacPkt contains the information needed by *BasePhyLayer* when constructing the AirFrame to send to the channel. Right now it only contains the Signal initialized by the *BaseMacLayer*. See 2.3 for how the Signal is created by the *BaseMacLayer*.

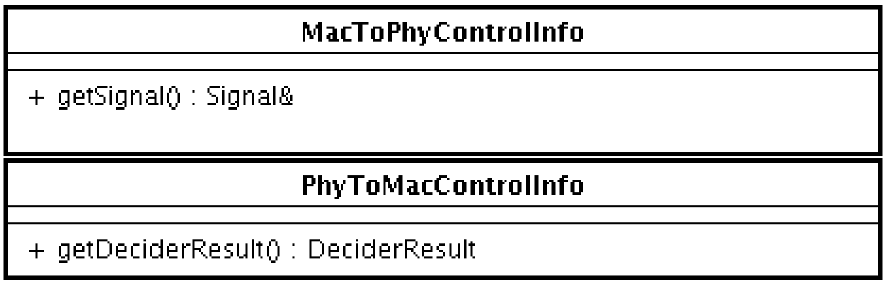

Figure 9: MacToPhyControlInfo interface

BasePhyLayer adds further information to the Signal and is responsible for creating and initializing the AirFrame and attaching the Signal to it. For detailed arrangement of information in Signal and AirFrame see 2.7. When the AirFrame is complete and sent, *BasePhyLayer* schedules a TX_OVER message to the *BaseMacLayer* (via control-message).

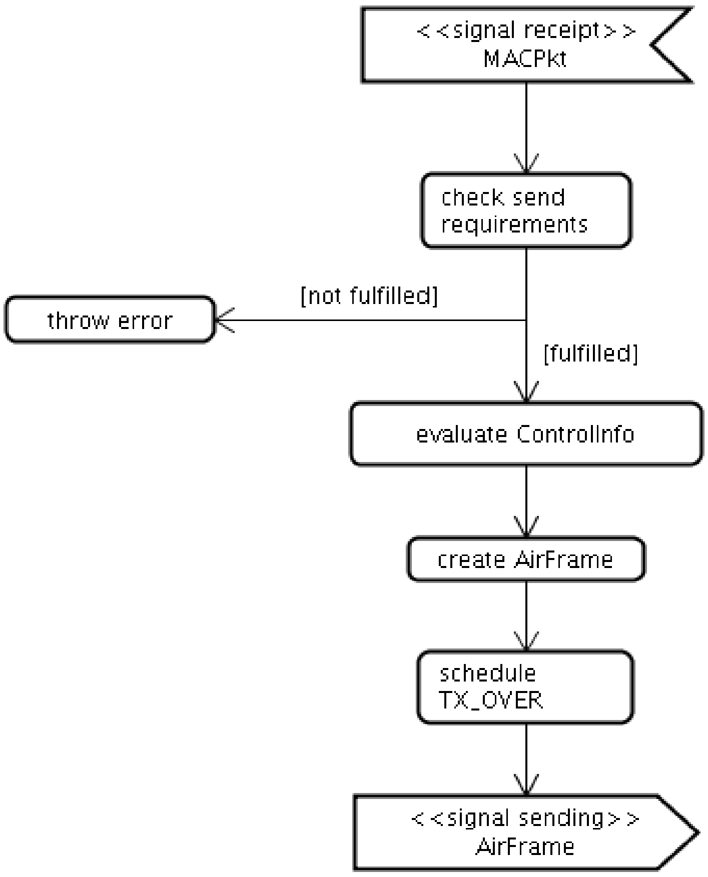

Figure 10: sending process

2.9 Receiving and processing an AirFrame

On arrival of an AirFrame *BasePhyLayer*:

1. applies AnalogueModels to the corresponding Signal$^{(\text{req. } 1.5)}$,
2. receives the AirFrame$^{(\text{req. } 1.5)}$.

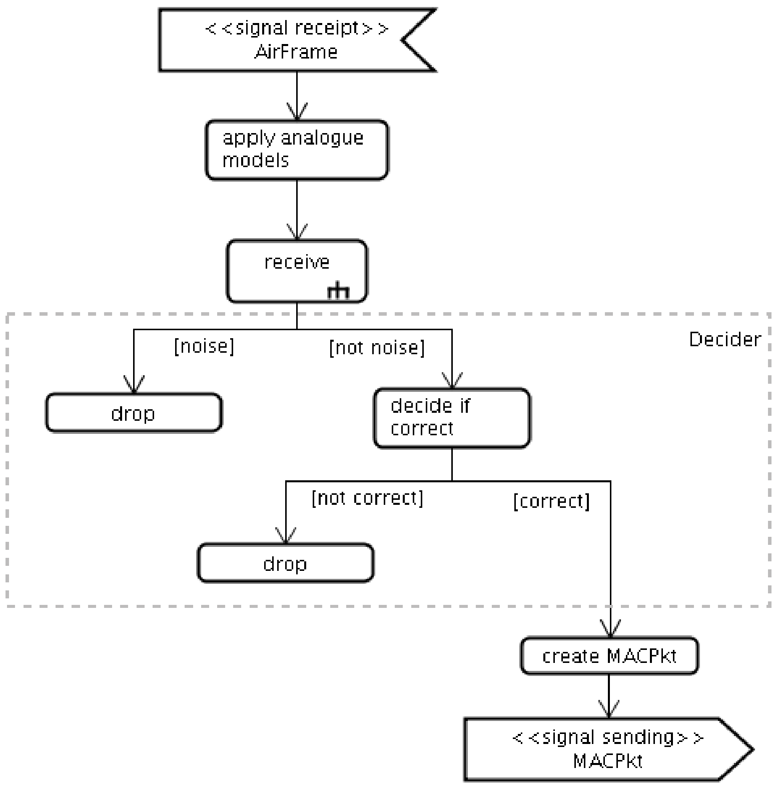

Figure 11: receiving process

The receiving process works as follows: In general, time intervals during reception are simulated by scheduling the AirFrame accordingly.

stage 0:
An optional propagation delay is simulated by updating the starting time of the Signal$^{(\text{req. } 1.5)}$ according to the delay and scheduling the AirFrame to the reception start point.

stage 1:
On reception start the Signal is given to the Decider for processing. The Decider returns a time point it wants to process the Signal again. This time point has to be before the end of the Signal otherwise an error is thrown.

stage 2:
The AirFrame is scheduled arbitrary times for the Decider processing method until it either returns a negative time point or the time point of the end of the Signal. In both cases the state is increased by one before the AirFrame is scheduled to its end. See 2.6 for more details on the Deciders Signal processing.

stage 3:
Finally reception is over and the AirFrame is completely received in reality.
Note: The Decider process is responsible for telling the BasePhyLayer to send a MacPkt to the BaseMacLayer.

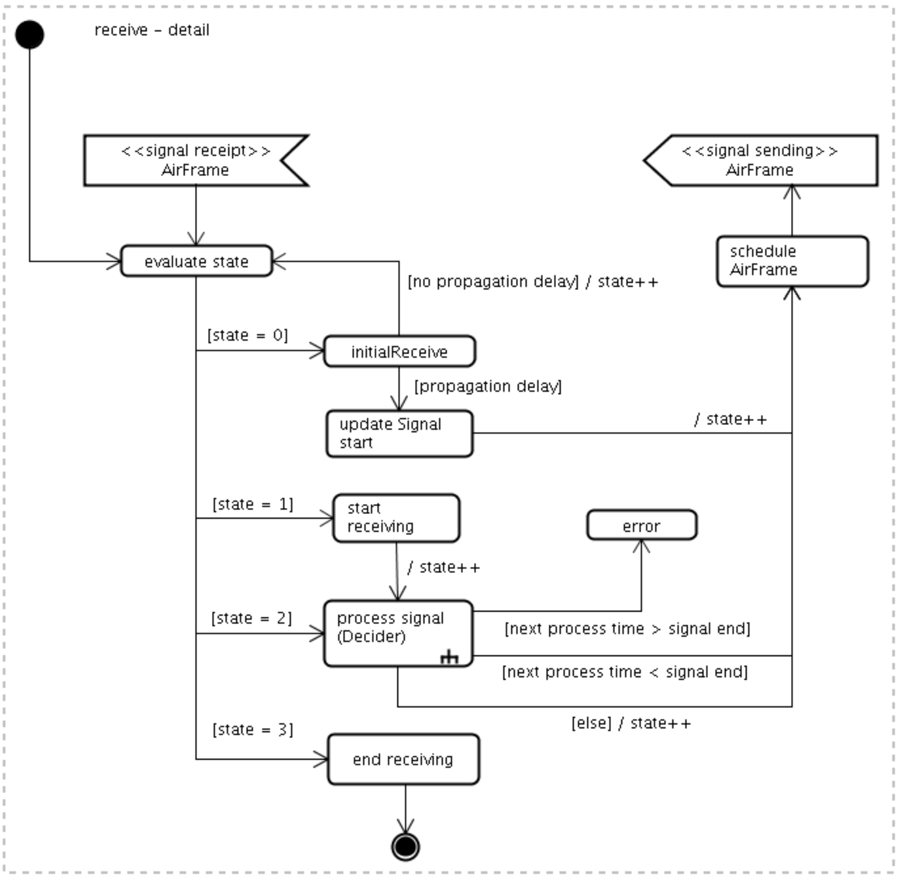

Figure 12: receive detail

2.10 the .ned-file

The .ned-file of the *BasePhyLayer* has the following parameters:

- usePropagationDelay as *boolean*$^{(\text{req. } 1.7)}$
- analogueModels as *XML*$^{(\text{req. } 1.7)(\text{req. } 1.7)}$

- decider as *XML*(req. 1.7)(req. 1.7)
- thermalNoise as *numeric const*(req. 1.7)
- sensitivity as *numeric const*(req. 1.7)
- maximal TX power as *numeric const*(req. 1.7)
- switchTimeRX as *numeric const*(req. 1.7)
- switchTimeTX as *numeric const*(req. 1.7)
- switchTimeSleep as as *numeric const*(req. 1.7)

The parameters ”analogueModels” and ”decider” store which AnalogueModels and which Decider are to be used, together with their parameters in XML format. The exact format still has to be declared!

2.11 signal details

The Signal class should be able to support multiple dimensions for arguments (time, channel, space, ...) and values (TX power, attenuation, bitrate). That means both sides of our signal function have to be extendable.
The following section explains how we achieve this.

2.11.1 multiple dimensions for values

Generally spoken a function can be implemented as a map from "Argument" to "Value". With inheritance we could basically use everything as a Value. If we assume that every Value of our Signals is in $\mathbb{R}^n$ we could implement it as a vector of doubles of size n.

2.11.2 multiple dimensions for arguments

While we could use basically everything as Value the Argument has to be distinguishable. Further a simple map with an Argument class as key would be hard to iterate reasonable and fast.
So instead of using one map from Argument to Value we realize multiple dimensions by using sub signals (see figure 13). We add a dimension of arguments by implementing a map from the argument value to another sub-signal. This structure has three advantages:

1. The Signal dimensions are easy to extend.
2. It provides a deterministic way to iterate over it.
3. And if every sub signal knows its dimension we can still use an Argument class (which would be, same as the Value, an extendable vector of doubles) for direct access to a specific Value of the Signal.

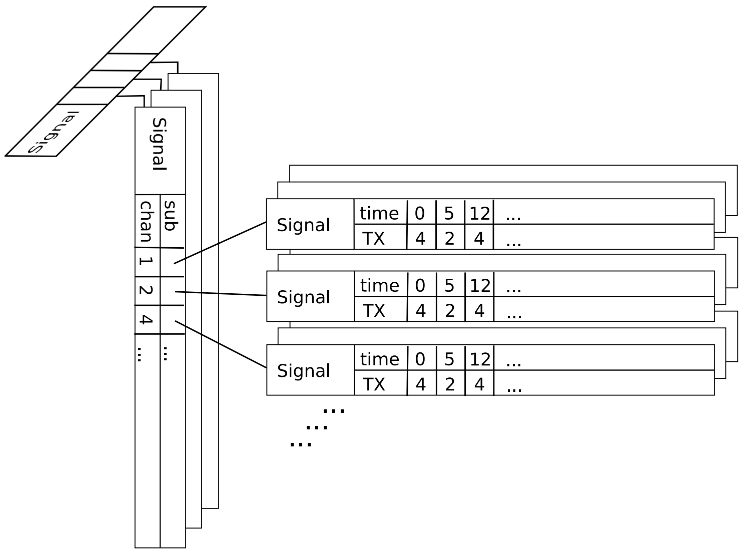

Figure 13: multi dimensional signals

2.11.3 creating and identifying Signals

Three things have to be considered:

1. We have three classes now (Signal, Argument and Value) which depends all on the type (meaning the dimensions) of the Signal. So we have to make sure to create the appropriate Arguments and Values for a Signal.
2. A receiving physical layer could receive different types of Signals during runtime, so we need a way to identify the type of a Signal.
3. The Signal types have to be globally known by each Physical Layer in the simulation.

We solve the first two problems by introducing a SignalPrototype class. If we want to create a new Signal type we create a new SignalPrototype with the parameters for the Signal type (which would be the dimensions). The Signal-Prototype then is able to create the appropriate Signals, Arguments and Values. It is also used as Identifier for the Signal type.
The last problem is solved by a simulation wide known SignalDatabase which maps a Signal type name ("MIMO signal" for example) to the associated Prototype. Every new SignalPrototype has to be registered by a unique name with the SignalDatabase.

2.11.4 detailed class diagram of Signal

The above decisions lead to the following class diagram:

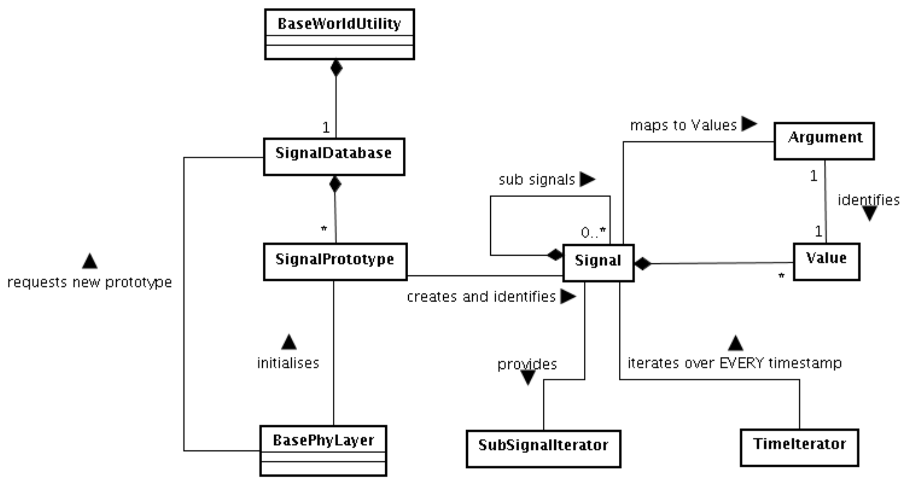

Figure 14: detailed class diagram

2.11.5 creation of new signal types

To assure that every new SignalPrototype is registered with the database we don't provide a public constructor for the SignalPrototype. If a physical layer wants to create a new prototype it has to ask the database if a prototype with the same name is already registered. If not it can ask the database to create a new one with the given name.

We assume that every Signal has at least time as argument dimension and TX power and attenuation as value dimensions. So these dimensions are already included in every new prototype.
Additional dimensions have to be registered by name with the prototype. The registration methods returns a unique sequential number for the registered dimension. This number is used later as a fast way to access the values of the dimension.
If every dimension is registered at the prototype one can create proper new Signals, Arguments and Values with this SignalPrototype.

Note: The first time the SignalPrototype is used to create a Signal, Argument or Value it is internally locked and no other dimensions can be registered.

The "isPrototypeFor()" method can be used to check if a Signal was created by this Prototype.

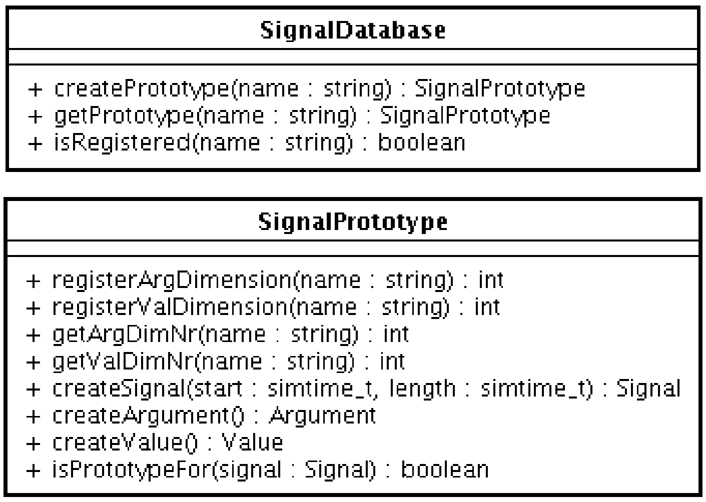
SignalDatabase
+ createPrototype(name : string) : SignalPrototype
+ getPrototype(name : string) : SignalPrototype
+ isRegistered(name : string) : boolean
SignalPrototype
+ registerArgDimension(name : string) : int
+ registerValDimension(name : string) : int
+ getArgDimNr(name : string) : int
+ getValDimNr(name : string) : int
+ createSignal(start : simtime_t, length : simtime_t) : Signal
+ createArgument() : Argument
+ createValue() : Value
+ isPrototypeFor(signal : Signal) : boolean

Figure 15: interface of Database and Prototype

2.11.6 accessing the Signal

The following class diagram shows the methods which the Signal provides **additionally** to those listed in figure 8.

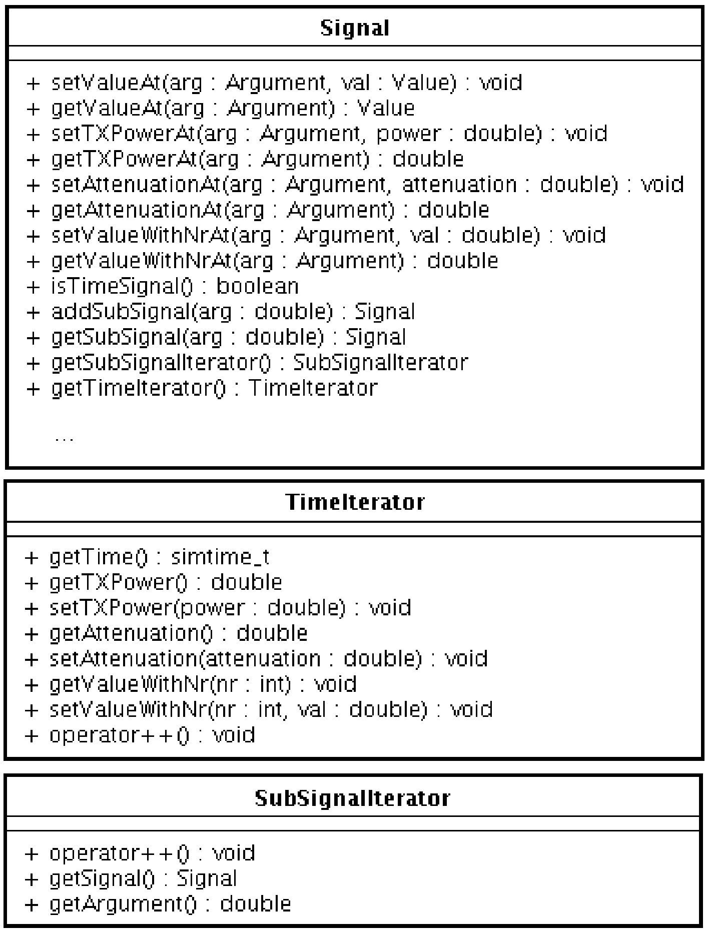
Signal
+ setValueAt(arg : Argument, val : Value) : void
+ getValueAt(arg : Argument) : Value
+ setTXPowerAt(arg : Argument, power : double) : void
+ getTXPowerAt(arg : Argument) : double
+ setAttenuationAt(arg : Argument, attenuation : double) : void
+ getAttenuationAt(arg : Argument) : double
+ setValueWithNrAt(arg : Argument, val : double) : void
+ getValueWithNrAt(arg : Argument) : double
+ isTimeSignal() : boolean
+ addSubSignal(arg : double) : Signal
+ getSubSignal(arg : double) : Signal
+ getSubSignalIterator() : SubSignalIterator
+ getTimeIterator() : TimeIterator
...
TimeIterator
+ getTime() : simtime_t
+ getTXPower() : double
+ setTXPower(power : double) : void
+ getAttenuation() : double
+ setAttenuation(attenuation : double) : void
+ getValueWithNr(nr : int) : void
+ setValueWithNr(nr : int, val : double) : void
+ operator++() : void
SubSignalIterator
+ operator++() : void
+ getSignal() : Signal
+ getArgument() : double

Figure 16: signal interface details

The TimeIterator still iterates over every time stamp of the Signal, independent from the dimension of the Signal. This way simple filters can be applied without knowledge of the exact type of the Signal.

The easiest way to access a specific value of the Signal, is to create an instance of the apropriate Argument (and Value, if you want to change it), initialize its dimensions with the wanted values and then use the "getValueAt()" and "setValueAt()" methods. Here one has to know that an entry for the given Argument actually exists, otherwise the Signal would have to interpolate the Value.

The SubSignalIterator can be used to iterate recursively over every pair of value and its associated sub signal of the dimension this Signal represents. The deepest

SubSignal will always be the time dimension which can be checked by "isTimeSignal()".

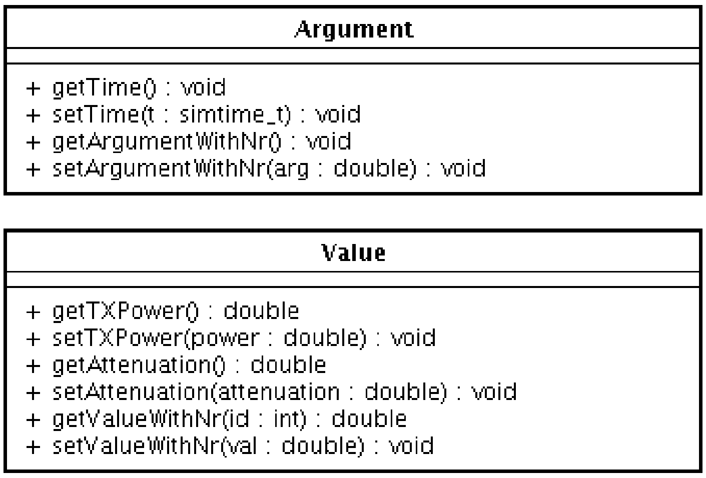
Argument
+ getTime() : void
+ setTime(t : simtime_t) : void
+ getArgumentWithNr() : void
+ setArgumentWithNr(arg : double) : void
Value
+ getTXPower() : double
+ setTXPower(power : double) : void
+ getAttenuation() : double
+ setAttenuation(attenuation : double) : void
+ getValueWithNr(id : int) : double
+ setValueWithNr(val : double) : void

Figure 17: interface for Argument and Value

The SignalPrototype associates every argument dimension name with a unique sequential number. The same holds for the value dimensions. These numbers can be used to access the values in Argument or Value with the "get*WithNr()" methods.
Because every Argument has a time dimension and every Value a TX power and an attenuation, the classes provide short cuts to these values.