NS-3 Reference Manual

Module: PhySim-WiFi

Jens Mittag, Stylianos Papanastasiou
jens.mittag@kit.edu, stylianos@gmail.com

PhySim-WiFi version: 1.2 25 April 2012

This is the reference manual for a detailed WiFi physical layer implementation called *PhySim-WiFi*, which models the signal processing logic of a transceiver down to the time sample level.

This document is written in GNU Texinfo and is maintained within the NS-3 module *physim-wifi* located under `src/contrib/physim-wifi` of the main NS-3 source tree. The authors of this module are Jens Mittag (`jens.mittag@kit.edu`) and Stylianos Papanastasiou (`stylianos@gmail.com`). If you have questions, either contact the NS-3 developers mailinglist (`ns-developers@isi.edu`) or contact the authors directly.

Short Contents

1 Introduction

This module contains a physical layer implementation of the OFDM-based IEEE 802.11 standard, more precisely, for the Orthogonal frequency division multiplexing (OFDM) PHY specification for the 5 GHz band[1]. It can be used as a drop-in replacement for the official `YansWifiPhy` implementation when higher simulation accuracy is required.

The NS-3 default physical layer, `YansWifiPhy`, implements a packet-level PHY model which abstracts channel effects on individual packet bits by using average bit-error rates w.r.t. signal-to-noise and interference ratios to determine whether a packet is successfully received. In contrast, the `PhySimWifiPhy` implementation performs all signal processing steps that a real transceiver would follow when decoding a frame. As such, individual bits are explicitly considered and detailed lower-layer techniques such as bit interleaving, forward error correction, OFDM modulation and so on are applied. The end result of accounting for these mechanisms is a detailed and accurate signal representation, allowing consideration of effects such as frequency- and time-selective fading as well as enabling evaluation of the impact of advanced physical layer signal processing algorithms on the performance of the whole network. Further, by modeling the physical layer at this granularity, existing and new wireless channel models can easily be implemented and plugged into the simulator, without the need to build empirical bit-error or packet-error rates. For additional information on the motivation of this work, consult the publications "Bridging the Gap between Physical Layer Emulation and Network Simulation" or "Enabling Accurate Cross-Layer PHY/MAC/NET Simulation Studies of Vehicular Communication Networks".

Figure 1.1 shows the conceptual architecture of the `PhySimWifiPhy` implementation, how it interacts with the existing WiFi MAC layer implementation and which sub-modules (i.e. signal processing modules) are used to simulate the frame construction and frame reception of a transceiver. In the following, the manual gives a basic overview of the frame transmission and reception process. Further details are then provided in Chapter 2 [Implementation], page 3.

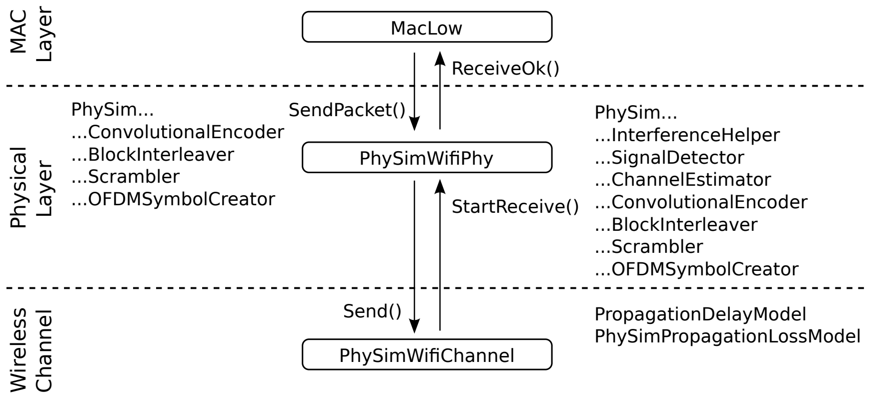

Figure 1.1: Architecture of the PhySimWifi implementation: how it connects to the existing WiFi MAC implementation and which sub-modules are used to simulate the frame construction and frame reception process.

[1] see Section 17 of the IEEE 802.11 (2007) standard

Whenever the MAC layer triggers a `SendPacket()` request on a `PhySimWifiPhy` instance, the frame construction process is started and a transition from packet-level to bit-level and from bit-level to signal-level is performed. To do so, the data bits of the packet are taken as an input (if given; if not, a random bit sequence that corresponds to the payload length is generated) and the functionality of several sub-modules is used to achieve the transformation. The necessary sub-modules are called `PhySimConvolutionalEncoder`, `PhySimBlockInterleaver`, `PhySimScrambler` and `PhySimOFDMSymbolCreator`, and they correspond to the transformations specified in the IEEE 802.11 standard for OFDM-based transmission. Section 2.1 [Frame Construction Process], page 3 of this manual elaborates further on the details of the frame construction process.

After the complex time samples, which constitute the packet, have been generated, the packet is passed down to the wireless channel (which is of type `PhySimWifiChannel`). The channel computes the propagation delay (using the existing `PropagationDelayModel` implementations), applies a propagation loss (using sub-classes of `PhySimPropagationLossModel`) and schedules a corresponding `StartReceive()` event at the receiving `PhySimWifiPhy`[2]. For further details on how a propagation loss model can manipulate the signal, look at Section 2.2 [Modeling the Wireless Channel Effects], page 5.

When the `StartReceive()` event expires, the incoming packet is first added to the `PhySimInterferenceHelper` module, which keeps track of all currently incoming frames. Afterwards, the reception process begins, depending on the current transmission and reception state of the receiving physical layer. During the reception process, two additional sub-modules are used, apart from the ones already mentioned in the transmission process, namely `PhySimSignalDetector` and `PhySimChannelEstimator`. Further implementation details are described in Section 2.3 [Frame Reception Process], page 6.

Note that the increase in simulation accuracy is accompanied by an increase in computional effort. Depending on the WiFi mode used, the number of network nodes simulated, the amount of packets and data bits transmitted and the used channel models, the simulation can be up to 1000 or 10000 times slower than the default `YansWifiPhy` implementation. In general, simulation requirements decrease with lower amounts of transmitted data, higher PHY data rates (transmission mode) and simpler channel models. A future version will include optional support for OpenCL, such that the signal processing parts can be parallelized, e.g. through the usage of recent GPGPUs with multiple compute units and cores (cf. "GPU-based Architectures and their Benefit for Accurate and Efficient Wireless Network Simulations" for initial performance results).

1.1 System Requirements

The following libraries are required to build and use the `PhySimWifiPhy` implementation:

- IT++ Signal Processing Library, Version 4.0.6 or higher

[2] The term 'receiving' might be misleading here, since the frame might arrive, but not be received, because the physical layer might not be able to detect the frame or synchronize to it.

2 Implementation

The following subsections contain detailed descriptions of the frame construction and reception processes, the implementation of the wireless channel and the interaction of the new PHY with higher layer functions such as MAC busy and idle state signaling.

2.1 Frame Construction Process

The frame construction process starts whenever a `SendPacket()` is initiated from higher layers, that is, in a typical case, from `MacLow`. Frame construction entails the transformation of individual bits in the packet into a sequence of complex time samples as well as the creation of a corresponding signal header and a preamble containing the OFDM short and long training symbols. The final constructed frame adheres to the frame format illustrated in Figure 2.1.

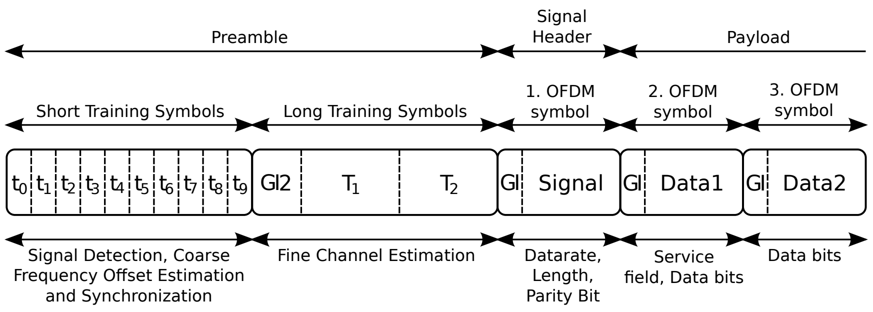

Figure 2.1: IEEE 802.11 PHY format for for the Orthogonal frequency division multiplexing (OFDM) PHY specification for the 5 GHz band.

In detail, frame construction consists of the following steps:

1. Get the payload of the packet via `packet->PeekData()`. The data is transformed to a vector of bits (`itpp::bvec`) and either reflects the real bits of the packet (if given), or a sequence of zero bits. The latter case does not lead to an immediately predictable bit pattern, however, since the bits are randomly scrambled later on in the construction process using a random seed.
2. The preamble of the packet is created using `PhySimWifiPhy::ConstructPreamble()`.
3. The signal header is created with BPSK modulation and a coding rate of 1/2 using `PhySimWifiPhy::ConstructSignalHeader(uint32_t length, const WifiMode mode)`.
4. The payload of the packet is encoded at the requested WiFi rate using `PhySimWifiPhy::ConstructData(const itpp::bvec& bits, const WifiMode mode)`.
5. The preamble, the signal header and the data symbols are concatenated into a single vector of complex time samples, i.e. into a `itpp::cvec`.
6. The energy of the samples produced in the above steps is normalized to unit power and then transmit power and transmitter antenna gain factors are applied to the samples.

7. All running `EndPreamble`, `EndHeader` and `EndRx` events are canceled, since it is the task of MAC to prevent transmissions when the receiver is busy (Section 2.5 [Integration with MAC Layer], page 8 elaborates on the details).
8. The effect of transmitter oscillator offsets, i.e. a slight frequency offset, is applied (to the time samples).
9. A `PhySimWifiPhyTag` is created to store the complex time samples and bundle other information that is used by the signal processing sub-modules (e.g. channel estimate values, transmit WiFi mode, transmitted data bits).
10. Physical layer state is changed through `PhySimWifiPhyStateHelper::SwitchToTx`.
11. The packet is passed on to `PhySimWifiChannel`.

Figure 2.2 describes the general signal processing steps that take place within `PhySimWifiPhy::ConstructData`. First, the input bits are scrambled by the `PhySimScrambler` using a randomly generated scrambling sequence in order to prevent long trails of consecutive 0s and 1s. In the second step, the `PhySimConvolutionalEncoder` adds forward error correction bits at a rate that depends on the requested WiFi mode (e.g. at rate 1/2 when using 6 Mbps in a 20 MHz channel). Afterwards, the bit sequence is divided into equally sized blocks, whereas each block has to contain exactly the right number of bits that will be constitute one OFDM symbol later (cf. NCBPS parameter of the IEEE 802.11 standard, Section 17).

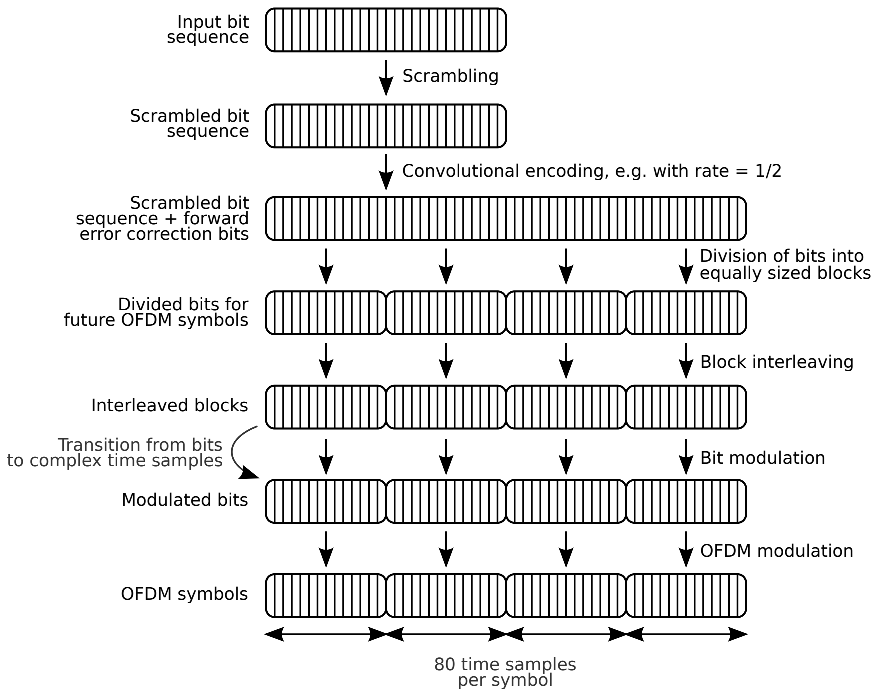
Input bit sequence
Scrambling
Scrambled bit sequence
Convolutional encoding, e.g. with rate = 1/2
Scrambled bit sequence + forward error correction bits
Division of bits into equally sized blocks
Divided bits for future OFDM symbols
Block interleaving
Interleaved blocks
Transition from bits to complex time samples
Bit modulation
Modulated bits
OFDM modulation
OFDM symbols
80 time samples per symbol

Figure 2.2: The signal processing steps that perform the transformation from bit-level to the final OFDM symbols at signal level.

For each block, the bits are mapped to the correct sub-carriers by the `PhySimBlockInterleaver` in order to avoid long runs of low reliability bits, then modulated according to the requested WiFi mode (i.e. either using BPSK, QPSK, 16-QAM or 64-QAM) and, finally, passed on to the OFDM modulator. The OFDM modulator inserts four pilot symbols at pre-defined sub-carrier positions; these can later be used by the receiver to track the channel characteristics over time. The last two steps are implemented within the `PhySimOFDMSymbolCreator` and the result of all the signal processing steps described above is a sequence of complex time samples, where each block of 80 time samples corresponds to one OFDM symbol.

In `PhySimWifiPhy::ConstructSignalHeader`, the same sequence of signal processing steps as above is used, except that there is no scrambling of bits and the simplest modulation scheme (BPSK) and a coding rate of 1/2 are used.

2.2 Modeling the Wireless Channel Effects

The wireless channel multiplexer, i.e. the entity that interconnects the `PhySimWifiPhy` instances, is implemented in subclasses of type `PhySimWifiChannel`. The multiplexer has to be configured to use instances of type `PhySimPropagationLossModel` and `PropagationDelayModel` in order to apply propagation loss effects (for pathloss, shadowing and fast fading) and propagation delays.

Currently, two channel multiplexer implementations are available, the classic `PhySimWifiUniformChannel` and a `PhySimWifiManualChannel`. The uniform multiplexer applies the same propagation and delay models to all wireless links. As such, all links will be exposed to the same channel effects (e.g. all links will experience a ThreeLogDistance pathloss and a Rayleigh fading). In comparison, the `PhySimWifiManualChannel` further allows to specify individual delay and propagation loss models for individual links. For instance, one can specify a default ThreeLogDistance pathloss to all links (with a given set of pathloss exponents), and refine the channel conditions between individual nodes by configuring a different pathloss model for those.

Independent of the used multiplexer, the PhySim WiFi module contains the following propagation loss models, which all operate on the complex time samples in order to reflect pathloss, shadowing or frequency- and time-selective fast-fading effects:

- `PhySimFriisSpacePropagationLoss`, in order to apply the well-known Friis Space propagation model
- `PhySimConstantPropagationLoss`, in order to apply a constant propagation loss (in dB)
- `PhySimLogDistancePropagationLoss`, in order to apply a logarithmic pathloss w.r.t. distance
- `PhySimThreeLogDistancePropagationLoss`, in order to use different pathloss exponents for different distance ranges
- `PhySimShadowingPropagationLoss`, in order to apply a shadowing effect with a Gaussian normal random variable
- `PhySimRicianPropagationLoss`, in order to apply a fast-fading Rician effect (or Rayleigh, if no LOS component is set) that takes also Doppler effects into account

- PhySimTappedDelayLinePropagationLoss, in order to use the multi-tap channel implementations that are included within IT++ (e.g. ITU and COST models)
- PhySimVehicularChannelPropagationLoss, in order to apply sophisticated channel effects, that are based on channel soundings in the 5.9 GHz domain using different environments (Urban Canyon, Suburban Street, Expressway) and different experiment setups (Oncoming and Same Direction)

For further information about the implemented channel models, interested readers may refer to the API Documentation, which outlines implementation details and gives references to relevant works in the literature.

2.3 Frame Reception Process

The frame reception process is divided into 4 events within PhySimWifiPhy, see Figure 2.3: the initial PhySimWifiPhy::StartReceive event that indicates that the first complex time sample has propagated through the channel and has arrived at the receiver, the PhySimWifiPhy::EndPreamble event, that indicates that all complex time samples that represent the frame preamble with all of its training symbols have arrived at the receiver, the PhySimWifiPhy::EndHeader event which expires once all complex time samples that represent the signal header have arrived, and finally the PhySimWifiPhy::EndRx event, which indicates when the whole frame has finally arrived.

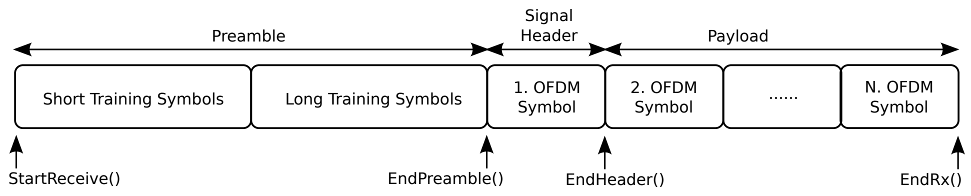

Figure 2.3: The frame reception process and its division into four distinct events.

At PhySimWifiPhy::StartReceive, the packet is added to PhySimInterferenceHelper in order to keep track of the interference added by the arriving packet. Further, depending on the current PHY state, that is only if the physical layer is currently in IDLE or CCA_BUSY state, a new PhySimWifiPhy::EndPreamble event is scheduled and it is checked whether a new CCA_BUSY phase has to be started (e.g. because the cumulative signal energy at the received now exceeds the configured CcaBusyThreshold). Since v1.1, also packet capture is supported, i.e. an PhySimWifiPhy::EndPreamble event is also scheduled if the physical layer is currently in SYNC or RX state and if the signal-to-interference noise ratio is greater or equal than 8 dB. For further information regarding packet capture and how it works, please take a look at "An Experimental Study on the Capture Effect in 802.11a Networks".

At PhySimWifiPhy::EndPreamble, the PhySimSignalDetector module is used to check whether the repeating pattern of the short training symbols can be detected or not. If it can be detected and if the signal-to-interference noise ratio (SINR) is greater than 4 dB and if the physical layer is either in CCA_BUSY, IDLE or SYNC state, it is then assumed that the receiver can lock on to that frame. The 4 dB requirement is introduced in order to distinguish multiple preambles (that would be the case if the receiver is in SYNC state and two preambles are overlapping in time). Then the receiver performs an initial channel estimation also takes place through the PhySimChannelEstimator module. Afterwards, a

`PhySimWifiPhy::Endheader` event is scheduled and the physical layer state is changed to `SYNC`.

Note that for all events the cumulative time samples of all arriving packets are used as input for the signal processing. Contrary to v1.0, the signal detection mechanism does not process more than the 320 samples of the preamble anymore, since this adds only additional computation time while not increasing the accuracy of the simulation.

At `PhySimWifiPhy::EndHeader`, the initial channel estimate is applied to all time samples. Afterwards, the header is decoded by applying the reverse signal processing steps of the frame construction process. The result is a bit vector that contains the SIGNAL header of the packet. The header is then examined to determine the modulation and coding rate used for the rest of the frame, the frame length, and the parity bit. Only if all values in the header are plausible, and the physical layer is currently in `SYNC` state, a `PhySimWifiPhy::EndRx` event is scheduled after the expected end of the packet. In addition the physical layer state is changed to `RX`. Please note, that contrary to v1.0, running `PhySimWifiPhy::EndPreamble` events are not cancelled any more, since this would disable the packet capture feature.

At `PhySimWifiPhy::EndRx`, the data symbols are decoded. Again, the initial channel estimate is applied to all time samples. Afterwards, OFDM and bit modulation are reversed and the forward error correction bits are used to correct possible errors. Then, the initial state of the scrambler at the transmitter is recovered and used to re-arrange the bits into their original order. And if the decoded data bits are identical to the transmitted data bits in the end, the reception is considered to be successful, and the physical layer state is changed either to `IDLE` or `CCA_BUSY`, depending on whether the cumulative signal strength is above or below the configured CcaBusyThreshold.

2.4 Physical Layer State Machine

Based on the frame format and the events of the reception process, the physical layer distinguishes between 5 different states, see Figure 2.4: the `TX` state, which is active when the physical layer is transmitting a packet, the `IDLE` state, which is active whenever no reception process is going on and the whenever the cumulative signal strength is below the CcaModelThreshold, the `CCA_BUSY` state, which is similar to `IDLE`, except that the signal strength is above CcaModelThreshold, the `SYNC` state, which reflects the situation that a preamble has been detected and locked on to, and finally the `RX` state, which is active whenever the receiver is decoding the payload of a packet.

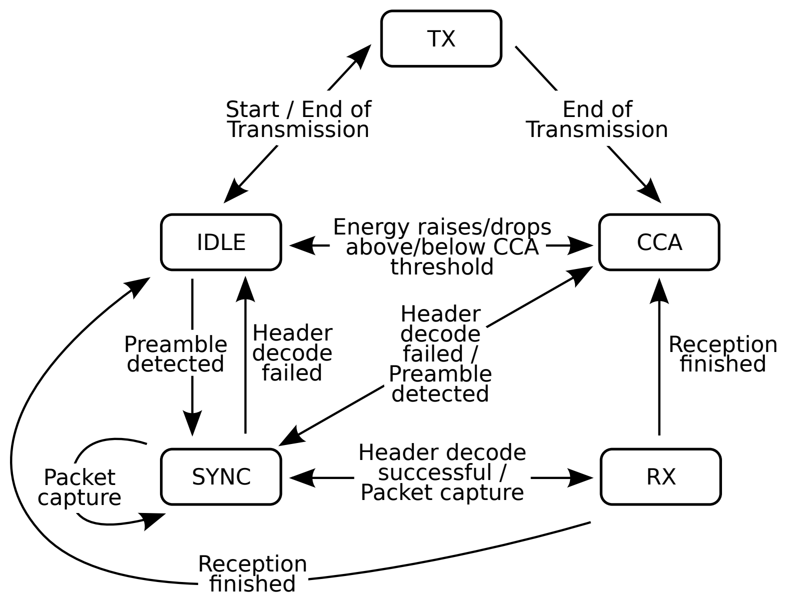
TX
Start / End of Transmission
End of Transmission
IDLE
Energy raises/drops above/below CCA threshold
CCA
Preamble detected
Header decode failed
Header decode failed / Preamble detected
Reception finished
Packet capture
SYNC
Header decode successful / Packet capture
RX
Reception finished

Figure 2.4: The state machine of the physical layer implementation and the allowed transitions between those states.

The transitions of the physical layer state machine are implemented through `PhySimWifiPhyStateHelper`, which is responsible for transition checks (e.g. if a transition from the current state into a new state is allowed) and the notification of other layers, e.g. the MAC[1].

2.5 Integration with MAC Layer

The physical layer provides feedback to the MAC layer whenever the receiver is in `TX`, `RX` or `CCA_BUSY` state, in order to enable the CSMA mechanism of IEEE 802.11. The notification is performed in `PhySimWifiPhyStateHelper`, whereas `PhySimWifiPhyStateHelper::NotifyRxStart` is used to implement the virtual carrier sensing functionality, while `PhySimWifiPhyStateHelper::NotifyMaybeCcaBusyStart` is used for the physical carrier sensing counterpart.

In order to trigger the notifications for physical carrier sensing correctly, `PhySimWifiPhy` contains two methods to detect the start of the next `CCA_BUSY` phase, start the detected phase and end an already running `CCA_BUSY` phase. These are, respectively, `PhySimWifiPhy::CheckForNextCcaBusyStart`, `PhySimWifiPhy::StartCcaBusy` and `PhySimWifiPhy::EndCcaBusy`. `CheckForNextCcaBusyStart` is called whenever the receiver needs to decide whether to switch to the `IDLE` or `CCA_BUSY` state; specifically, it is called after a `PhySimWifiPhy::EndRx` event, a failed `PhySimWifiPhy::EndPreamble` or a failed `PhySimWifiPhy::EndHeader` event, and whenever there is a need to check whether to stay `IDLE` or not, as outlined in `PhySimWifiPhy::StartReceive`.

[1] Please note that the state machine presented in "Bridging the Gap between Physical Layer Emulation and Network Simulation" is incorrect and allows (by mistake) a transition from `CCA_BUSY` to `TX`.

Checking whether to turn to a `CCA_BUSY` state is made on a per 80 time samples block level by computing the cumulative signal strength of such blocks (cf. `PhySimHelper::GetOFDMSymbolSignalStrength`. Once the signal strength exceeds the CcaModelThreshold, the state is switched to `CCA_BUSY`.

3 Examples and Usage

3.1 How to use `PhySimWifiPhy` instead of `YansWifiPhy`

When using the `YansWifiPhy` implementation, one would use something similar to the following example code to configure and set everything up.

```
#include "ns3/wifi-module.h"

YansWifiChannelHelper wifiChannel;
wifiChannel.AddPropagationLoss("ns3::FixedRssLossModel");
wifiChannel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");

YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default();
wifiPhy.SetChannel(wifiChannel.Create());

WifiHelper wifi = WifiHelper::Default();
wifi.SetStandard(WIFI_PHY_STANDARD_80211a);
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifiMac.SetType ("ns3::AdhocWifiMac");
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
  "DataMode", StringValue ("OfdmRate6Mbps"),
  "NonUnicastMode", StringValue ("OfdmRate6Mbps"));

NodeContainer nodes;
nodes.Create(2);

wifi.Install (wifiPhy, wifiMac, nodes);
```

In order to use the `PhySimWifiPhy` implementation instead of `YansWifiPhy`, one has to replace the `YansWifiChannelHelper` and `YansWifiPhyHelper` classes with `PhySimWifiChannelHelper` and `PhySimWifiPhyHelper` only, see below.

```
#include "ns3/wifi-module.h"
#include "ns3/physim-wifi-module.h"

PhySimWifiChannelHelper wifiChannel;
wifiChannel.AddPropagationLoss("ns3::PhySimConstantPropagationLoss");
wifiChannel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");

PhySimWifiPhyHelper wifiPhy = PhySimWifiPhyHelper::Default();
wifiPhy.SetChannel(wifiChannel.Create());

WifiHelper wifi = WifiHelper::Default();
wifi.SetStandard(WIFI_PHY_STANDARD_80211a);
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifiMac.SetType ("ns3::AdhocWifiMac");
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
  "DataMode", StringValue ("OfdmRate6Mbps"),
  "NonUnicastMode", StringValue ("OfdmRate6Mbps"));

NodeContainer nodes;
nodes.Create(2);
```

```
  wifi.Install (wifiPhy, wifiMac, nodes);
```

3.2 How to trace events from `PhySimWifiPhy`

The whole implementation can act as a trace source at the occurence of different events. The most interesting and important events are probably the ones published in PhySimWifiPhy itself.

```
  void PhyTxTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag);
  void PhyStartRxTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag);
  void PhyStartRxErrorTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag, enum PhySimWifiPhy::ErrorReason reason);
  void PhyEnergyDetectionFailedTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag);

  void PhyRxOkTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag);
  void PhyHeaderOkTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag);
  void PhyPreambleOkTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag);
  void PhyRxErrorTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag, enum PhySimWifiPhy::ErrorReason reason);
  void PhyHeaderErrorTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag, enum PhySimWifiPhy::ErrorReason reason);
  void PhyPreambleErrorTrace (std::string context, Ptr<const Packet> p,
                   Ptr<const PhySimWifiPhyTag> tag, enum PhySimWifiPhy::ErrorReason reason);
  void PhyCcaBusyStart (std::string context, Ptr<const NetDevice> device, Time duration);

  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/Tx",
                   MakeCallback(&PhyTxTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/StartRx",
                   MakeCallback(&PhyStartRxTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/StartRxError",
                   MakeCallback(&PhyStartRxErrorTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/EnergyDetectionFailed",
                   MakeCallback(&PhyEnergyDetectionFailedTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/RxOk",
                   MakeCallback(&PhyRxOkTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/HeaderOk",
                   MakeCallback(&PhyHeaderOkTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PreambleOk",
                   MakeCallback(&PhyPreambleOkTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/RxError",
                   MakeCallback(&PhyRxErrorTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/HeaderError",
                   MakeCallback(&PhyHeaderErrorTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PreambleError",
                   MakeCallback(&PhyPreambleErrorTrace) );
  Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/CcaBusyStart",
                   MakeCallback(&PhyCcaBusyStart) );
```

Apart from the above, there is also a trace source available in `PhySimWifiPhyStateHelper` which allows the user to be notified about all the physical layer state changes and transitions.

```
void StateLogger (std::string context, Ptr<NetDevice>, Time start, Time end ,
                  enum PhySimWifiPhy::State state);
Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/State/State",
                 MakeCallback(&StateLogger));
```

3.3 How to change the signal processing configuration

The signal processing modules offer quite a few configuration parameters. For instance, the correlation threshold or the correlation technique of the signal detector can be set through the NS-3 attribute system. A user may also enable or disable soft decision decoding, oscillator effects, random scrambler initialization and other features. Since all these parameters are documented within the source code and exposed through the doxygen generated API documentation, this manual refers to the API documentation for a detailed description of all those parameters.

However, there is a small issue in the way NS-3 configures the Wifi standard, which prevents the proper change of default attributes after setting the physical layer to a specific IEEE standard. For instance if you use the following code to first configure `PhySimWifiPhy` to reflect a IEEE 802.11a setup and then want to change the CCA threshold from the default -62 dBm to -82 dBm, it won't work. The reason is that the attribute system does not apply the changes in chronological order.

```
WifiHelper wifi = WifiHelper::Default();
wifi.SetStandard(WIFI_PHY_STANDARD_80211a);
Config::SetDefault ("ns3::PhySimWifiPhy::CcaModelThreshold", DoubleValue (-82.0));
```

The proper way of doing this would be to not change the default value for the object instantiation, but to change the value after the physical layer has been created by using the corresponding `Config::Set` call, see blow.

```
WifiHelper wifi = WifiHelper::Default();
wifi.SetStandard(WIFI_PHY_STANDARD_80211a);
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/CcaModelThreshold",
             DoubleValue (-82.0));
```

This issue is existing for the `PhySimWifiPhy` attributes `CcaModelThreshold`, `Frequency`, `TxCenterFrequencyTolerance` and `SymbolTime`.

4 Additional Remarks

The current implementation is already stable and all the OFDM based communication modes are working and validated, either against the reference values of the example in Annex G of the IEEE 802.11 (2007) standard, against the CMU channel emulator testbed at Chalmers (see the publication mentioned in the beginning of this manual), or against theoretical results and curves. However, the current implementation has a few minor missing features, that will be added in a following up release:

- No multi-channel operation support so far
- It is possible to mix different data rates, as long as they share the same channel bandwidth. However, multiple (and overlapping) channels using different channel bandwidths (e.g. 10 and 20 MHz channels) are not supported, since different channel bandwidths imply different complex sample durations, which are not currently handled by the `PhySimInterferenceHelper`.

5 Changelog

5.1 Changes from v1.1 to v1.2

- Fix of an equation error in `PhySimLogDistancePropagationLoss::DoCalcRxPower`: instead of calculating `log10 (m_distance / m_referenceDistance)`, a `log10 (m_referenceDistance / m_distance)` was used in v1.1.
- An implementation of the method `PhySimInterferenceHelper::GetNoiseFloorDbm` was added.
- The attributes of `ns3::PhySimShadowingPropagationLoss` were renamed to `StandardDeviation` and `MinimumDistance`.
- Added configuration of IEEE 802.11g modes in `PhySimWifiPhy::ConfigureStandard`.
- Update in `PhySimWifiPhy::StartReceive` to properly handle the case of packet capture during `SYNCING` and `RX` states: now we notify a RxDrop in case that capture is not successful and tag the packet as being captured if it is successful.
- More time samples are used for preamble detection in `PhySimWifiPhy::EndPreamble`.
- If a packet is captured in `PhySimWifiPhy::EndPreamble`, it is tagged as being captured (using the `PhySimWifiPhyTag` object).
- A rare but significant bug in `PhySimWifiPhy::EndPreamble` has been fixed: so far, running EndCcaBusy events have not been canceled in case of successful preamble detection, which is, however, required to avoid unallowed state transitions.
- Bugfix in `PhySimWifiPhy::EndHeader`: it is necessary to check for a new CcaBusy period in case header decoding did not succeed.

5.2 Changes from v1.0 to v1.1

- A bug in Payload and Overall SINR Computation which let the whole simulation crash was fixed.
- The frequency offsets are modeled through a triangular variable instead of a uniform variable and have been moved from `PhySimWifiPhy::SendPacket` to `PhySimWifiPhy::StartReceive` (thanks to Michele Segata). In order to generate the frequency offset between sender and receiver, a triangular random variable is used. This choice derives from the following formulation. The frequency offset between sender and receiver is the difference between the *central* frequencies of the two wireless card, so $F_{off} = F_{tx} - F_{rx} = F_0 + Err_{tx} - (F_0 + Err_{rx}) = Err_{tx} - Err_{rx}$ where F_tx and F_rx are the frequencies of sender and receiver respectively, F_0 the center frequency (e.g., 5.9 GHz for 802.11p), and Err_tx and Err_rx the errors in frequency of sender and receiver respectively. Assuming Err_tx and Err_rx being two uniformly distributed random variables between -1 and 1 (multiplied by the offset tolerance as mandated by the standard like, e.g., 20 ppm for 20 MHz channel), the result of their difference is a triangular distribution between -2 and 2. Actually, the real distribution of central frequencies is generally not uniform, but it should depend on the technology used. We could expect a more concentrated distribution for modern devices. Uniform distribution is a worst-case scenario, but in the absence

of vendor-provided distributions this is the best we can do and it is good to test feasibility.

- Signal detector was improved and a default correlation threshold of 0.85 is configured. This translates to a minimum of 5db SNR in order to correctly detect the preamble. Please note that this only works in the absence of interference. In case of interference, an additional SINR check of 4dB is introduced to distinguish overlapping preambles.
- The accuracy of the SpeedOfLight value/attribute in `ConstantSpeedPropagationDelayModel` was increased.
- New trace sources have been added: StartRx, StartRxError and EnergyDetection-Failed. Furthermore, the trace sources PreambleError, HeaderError and RxError have been extended to include also an error reason. The reasons are defined `enum PhySimWifiPhy::ErrorReason`
- Frequency tolerance for 802.11p has been set to 20ppm again.
- A memory leak in `PhySimWifiPhy::EndRx` was resolved
- The `PhySimInterferenceHelper` class has now an optimization parameter/attribute called `MaximumPacketDuration` that can be used to reduce the amount of memory/history. This can be useful if the maximum packet duration is known prior to simulation, and memory consumption shall be reduced.
- A vector out of bound error in `PhySimWifiPhy::EndPreamble` was fixed.
- A rare bug in `PhySimInterferenceHelper::GetBackgroundNoise`, in particular within case 3 of this method, was fixed.
- Two bugs in the computation of V2V_EXPRESSWAY_ONCOMING and V2V_URBAN_CANYON_ONCOMING have been fixed.
- Packet capture capabilities have been added.