GSoC 2024 Plan

5G NR Module Benchmark and Analysis for Distinct Channel Models

João Albuquerque

Federal University of Pará, Telecommunications, Automation, and Electronics Research and Development Center (LASSE)

PROJECT MILESTONE:

Brief introduction:

Currently, 5G-Lena (v-3.0) uses different scenarios to be configured by the user for large-scale and small-scale fading parameters to be computed (e.g., propagation loss and 3GPP channel, using the RMA scenario). This configuration can be done using a manual definition or an API class. For manual operation, the user can configure the following:

```
                                                                    *
std::vector<std::reference_wrapper<std::unique_ptr<BandwidthPartInfo> >
> bwps;
     *    std::unique_ptr<BandwidthPartInfo> bwpi (new BandwidthPartInfo
(bwpId, centralFrequency,
                    *     bandwidth));     auto     spectrumChannel     =
CreateObject<MultiModelSpectrumChannel> (); auto
    * propagationLoss = CreateObject<FriisPropagationLossModel> ();
     *   propagationLoss->SetAttributeFailSafe ("Frequency," DoubleValue
(centralFrequencyBand1));
    *   spectrumChannel->AddPropagationLossModel (propagationLoss);
    *   bwpi->m_channel = spectrumChannel;
    *   bwps.push_back(bwpi);
```

In that case, the user is supposed to create the propagation and spectrum objects and then configure them outside of any API. On the other hand, the user can use a pre-defined API class (cc-bwp-helper) to ensure the correct configuration and initialization of the band; this is a more standard approach. To achieve this, the user needs to create a struct that holds all the current information of the band {Central frequency, Bandwidth, Number of component carriers, Scenario}. After this, the user can choose between `CreateOperationBandContiguousCc()` or `CreateOperationBandNonContiguousCc()`, using that struct that holds all the necessary information. These methods return the band, which can be initialized now using the `InitializeOperationBand()` *(in nr-helper)*. The initialized band method, as the name says, initializes the propagation, spectrum, and condition of the band that will be installed after in the UE and the gNB.

Limitations:

We can only initialize a 3GPP scenario using the API class for channel condition, propagation, and spectrum. However, ns-3 enables 5G-Lena to use distinct channel models (e.g., Fluctuating Two Ray (FTR), developed by Matteo Pagin in GSoC 2023), which is not used in the current version of 5G-Lena. Also, 3GPP has different features, which can scale with some parameters (e.g., number of antenna elements) that may cause more time duration for a simulation. But, sometimes, the user does not care about the channel, if it is calibrated or even trustworthy; perhaps they just want to evaluate another content that is not necessarily related to the physical simulation part.

In the newest version, 5G-Lena supports single-user multiple inputs and multiple outputs (su-MIMO). However, after some modifications, the NR module expects any implemented channel model to have a (spectrum) channel matrix. The problem is that some channel models do not calculate it, and NR cannot deal with the absence of this parameter.

Furthermore, the newest version of ns-3 (3.42) implemented new 3GPP scenarios related to the non-terrestrial network (NTN), which is not supported at the current moment in the 5G-Lena repo.

Modifications:

As described previously, in the standard code of 5G-Lena examples and tests, we use APIs to create, configure, and initialize the band. This project aims to modify these APIs to support the configuration of distinct channel models. The modifications will happen in the nr-helper class, which initializes the band. The idea is to extend the class to enable users to initialize other channel models (e.g., FTR and NYUSIM, developed by New York University (NYU)). Moreover, some modifications will be made in the cc-bwp-helper (ccBwpCreator class responsible for the band configuration). In this class, I propose another parameter that will be configured in the current band, the channel model. With this, the user can mix scenarios with the desired channel model, asserting if the channel model supports the chosen scenario.

Also, to support channel models that do not compute a channel matrix, I propose the calculation of the spectrum channel matrix inside the class related to the NR spectrum PHY (nr-spectrum-phy), which will be responsible for checking if the desired channel model has it if it does not have, then we create one, following the correct calculation. Furthermore, after finalizing all band configurations, creations, and initializations, we propose adding the already developed NTN scenarios in the current version of ns-3 for use in 5G-Lena.

Community Bonding Period (Q&A):

First meeting (09/05/2024):

A. Review the final proposal:

a. Align what we want and find the best way to execute and code.
    i. **Ans**: Next week, we will discuss my proposal and milestones with them.
b. Do we continue with the example and the test of the proposal? Let's define it later.
c. Can I change the src files of the ns-3 (e.g., two-ray-spectrum-propagation-loss.*)?
    i. **Ans**: Yes, it is better to work with the interfaces, but you have to take care about changing and adding a lot of code in each one

B. Parallel tasks:
    a. The issue to solve and help with something:
        i. **Ans**: Be engaged with the community (Zulip chat or ns-3 Google group).
C. Documentation (What do I have to know about…):
    a. The ns-3 spectrum class?
    b. The ns-3 propagation class?
    c. the NR module (nr-helper, cc-BWP-helper)
D. Talk about the best way to follow the main repository of NR and ns-3. Currently, I am following:
    a. nr: https://gitlab.com/cttc-lena/nr/-/blob/master/doc/development-guidelines.md
    b. ns-3:
        i. https://www.nsnam.org/docs/contributing/html/index.html
        ii. https://www.nsnam.org/docs/contributing/html/coding-style.html#coding-style
        iii. **Ans**: I only need to memorialize some code styles, rules, etc. Just make sure to verify e
E. Some recommended a book about MIMO that focuses on **channel modeling.**
    a. Ans: We will not need it.
F. Review my scratch (code part).
    a. **Ans**: Create a new branch to show my scratch.

Second meeting (17/05/2024):

A. Review the milestones on the wiki page. Do I have to change something?
   a. **Ans:** We did it now.

B. Channel Impulse Response (CIR) equation for a fast-fading model:

$$
\begin{aligned}
H_{u,s}(t,\tau) = & \sum_{n=1}^{N} \sqrt{\frac{P_n}{M}} \sum_{m=1}^{M} \bar{\mathbf{F}}_{rx}(\theta_{n,m}^{A}, \phi_{n,m}^{A}) \\
& \times \begin{bmatrix} e^{j\Phi_{n,m}^{\theta,\theta}} & \sqrt{K_{n,m}^{-1}}\, e^{j\Phi_{n,m}^{\theta,\phi}} \\ \sqrt{K_{n,m}^{-1}}\, e^{j\Phi_{n,m}^{\phi,\theta}} & e^{j\Phi_{n,m}^{\phi,\phi}} \end{bmatrix} \\
& \times \bar{\mathbf{F}}_{tx}(\theta_{n,m}^{D}, \phi_{n,m}^{D}) \\
& \times e^{j\mathbf{k}_{rx,n,m}^{T}\mathbf{d}_{rx,u}} e^{j\mathbf{k}_{tx,n,m}^{T}\mathbf{d}_{tx,s}} \\
& \times e^{j2\pi\nu_{n,m}t}\delta(\tau - \tau_n)
\end{aligned}
\qquad (2)
$$

- The element field patterns are calculated based on the Angle of arrival[az, el] (AoA) in the RX node and the Angle of Departure[az, el] (AoD) in the TX node.
- The array response (e^jk*d) needs the AoA and AoD too.
- The complex path gain is already calculated in GetFtrFastFading(), right?

$$
V_r = \sum_{n=1}^{N} \sqrt{\zeta} V_n \exp(j\phi_n) + X + jY, \qquad (2)
$$

```
// Compute the channel response by combining the above terms
std::complex<double> h = sqrtGamma * v1 * std::complex<double>(cos(phi1), sin(phi1)) +
                         sqrtGamma * v2 * std::complex<double>(cos(phi2), sin(phi2)) +
                         std::complex<double>(x, y);

return norm(h);
```

Where V_n is the specular component multiplied by a scalar factor and the complex value (std::complex<double>(x,y) is the diffuse received signal component.

Ans: The idea is to have something different from what we have in the 3gpp channel model, where we calculate using the equation shown. We can consider calculating individual channel responses for each rx PSD, but we are still determining its feasibility.

Third meeting (24/05/2024):

- Only finishing the Milestones

CODE PART:

WEEK 1:

Overview

The first question is how 5G-Lena commonly structures all the necessary information to initialize the bandwidth part (BWP). Look at cc-bwp-helper.*

```
/**
 * \ingroup helper
 * \brief Manages the correct creation of operation bands, component carriers and bandwidth parts
 *
 * This class can be used to setup in an easy way the operational bands needed
 * for a simple scenario. The first thing is to setup a simple configuration,
 * specified by the struct SimpleOperationBandConf. Then, this configuration can
 * be passed to CreateOperationBandContiguousCc.
 */
class CcBwpCreator
{
  public:
    /**
     * \ingroup helper
     * \brief Minimum configuration requirements for a OperationBand
     *
     * For instance, here is the simple configuration for a single operation band
     * at 28 GHz and 100 MHz of width:
     *
     * `CcBwpCreator::SimpleOperationBandConf bandConf1 (28e9, 100e6, 1,
     * BandwidthPartInfo::UMi_StreetCanyon);`
     *
     * The possible values of the scenario are depicted in BandwidthPartInfo
     * documentation.
     */
    struct SimpleOperationBandConf
    {
        /**
         * \brief Default constructor
         * \param centralFreq Central Frequency
         * \param channelBw Bandwidth
         * \param numCc number of component carriers in this operation band
         * \param scenario which 3gpp scenario path loss model will be installed for this band
         */
        SimpleOperationBandConf(double centralFreq = 28e9,
                                double channelBw = 400e6,
                                uint8_t numCc = 1,
                                BandwidthPartInfo::Scenario scenario = BandwidthPartInfo::RMa)
            : m_centralFrequency(centralFreq),
              m_channelBandwidth(channelBw),
              m_numCc(numCc),
              m_scenario(scenario)
        {
        }
```

This struct holds the information of the BWP, which are central frequency, bandwidth, number of component carriers (cc), and the scenario (used in the 3GPP channel model). In this class, 5G-Lena also has both:

```
/**
 * \brief Create an operation band with the CC specified
 * \param conf Minimum configuration
 *
 * Creates an operation band by splitting the available bandwidth into
 * equally-large contiguous carriers. Carriers will have common parameters like numerology.
 *
 *
 */
OperationBandInfo CreateOperationBandContiguousCc(const SimpleOperationBandConf& conf);


/**
 * \brief Creates an operation band with non-contiguous CC.
 *
 * \param configuration Minimum configuration for every CC.
 */
OperationBandInfo CreateOperationBandNonContiguousCc(
    const std::vector<SimpleOperationBandConf>& configuration);
```

Users can set a vector of *SimpleOperationBand* in ***CreateOperationBandNonContiguousCc*** to create the BWP with different attributes for each subband (I'm still determining). On the other hand, the ***CreateOperationBandContiguousCc*** splits all equally significant subbands.

CreateOperationBandContiguousCc:

Set:

- band Id;
- central frequency;
- channel bandwidth;
- lower frequency [centralfrequency - (bandwidth/2)];
- higher frequency [centralfrequency + (bandwidth/2)].
- max bandwidth of a cc

For each cc:

// Create the component carriers object

Create the cc (ccBandwidth, lower frequency, cc position, cc Iterator, numBwp, scenario)

// Initialize the created component carrier with the info below

Initialize it (cc, ccBandwidth, lower frequency of the band, cc position, cc ID)

// Initialize the created BWP with the info below, which sets the central frequency of the BWP, channel bandwidth, etc.

InitializeBWP(BWP object, bwpBandwidth, component carrier lower frequency, BWP number, BWP ID)

After configuring each component carrier aggregated to the current band, we can initialize it using the *InitializeOperationBand()*. Currently, 5G-Lena uses pre-defined functions mapped (with unordered_map) using the BWP scenario of each cc as a key. This map is a standard look-up table configuring the object factories (variables responsible for creating the correct class using the type ID) with the proper type ID. For example, if the user selects RMa, the scenario will be used as a key for that map. It will set the path loss and channel condition type ID (e.g., ns3::ThreeGppRmaPropagationLossModel and ns3::ThreeGppRmaChannelConditionModel), as defined in Fig. 1. This will happen until all the component carriers of the band been accomplished (initialized).

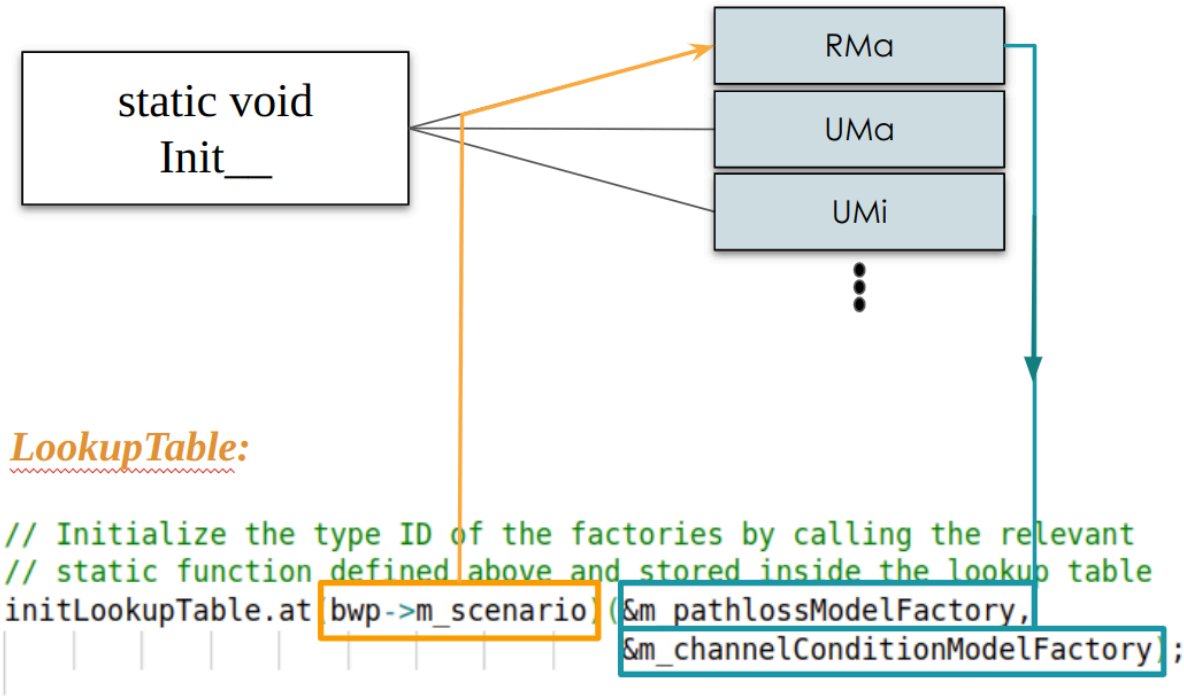

Fig. 1 - Configuring the path loss and channel condition type ID

The spectrum's type ID is already defined in the class's constructor and does not need to be set in the loop. Because we have only the 3GPP to be supported now, the method only creates the spectrum object (obviously, with the 3GPP type ID). It sets the right attributes for this class (scenario, central frequency, and channel condition). After all this, the band is installed in the gNB and the UE.

Presentation:

https://docs.google.com/presentation/d/18QROGyoguLpFE6mddkoVWXTJrMZ_S3GZCTb8GHphjp8/edit?usp=sharing

Conclusions and comments:

- This implementation works excellent but does not support other channel models; as described, only 3GPP is used
- mmWave + NYUSIM (**not explained here but in the presentation**) uses a similar approach, but they did not intend to create an API or a detailed implementation; they just made it work
- It is a good initial start, but this approach needs changes to support distinct channel models.

Week 2:

We discussed more about the first milestone and NYU interface implementation:

Presentation:

https://docs.google.com/presentation/d/1BGEXbTVMReKGh5ArjglZxAwyePU5ITeq7Y5SyTmq3Os/edit?usp=sharing

Week 3:

Milestone 1: Prepare 5G-Lena for different channel models that may not have spectrum channel matrix for su-MIMO

Problem Overview:

As mentioned in the project milestone section, one limitation of 5G-Lena at the current stage is the lack of support for channel models that are not matrix-based and do not have a (spectrum) channel matrix, non-su-mimo channel models. This happens because some features were developed during the upgrade of the NR module from 2.6 to 3.0; one is the computation of the interference signals from all out-of-cell interferers.

NR uses *CalcOutOfCellInterfCov*, *CalcCurrInterfCov*, *AddInterference*, and *ComputeSinr* to evaluate these interference signals, which will ensure that the signal parameters contain a (spectrum) channel matrix and will abort the simulation if it does not have one (e.g., `NS_ASSERT_MSG(firstSignal->spectrumChannelMatrix, "signal must have a channel matrix");`).

The designed and already implemented band configuration and initialization API will not generate an error because the 3GPP spectrum class creates the spectrum channel matrix. The problem appears when importing channel models that do not contain a channel matrix.

For example, trying to use a two-ray channel model in 5G-Lena in a simple communication between gNB and UE, I will receive an error in *CalcOutOfCellInterfCov*, after simulating other functions related to interference, as shown in Fig. 2.

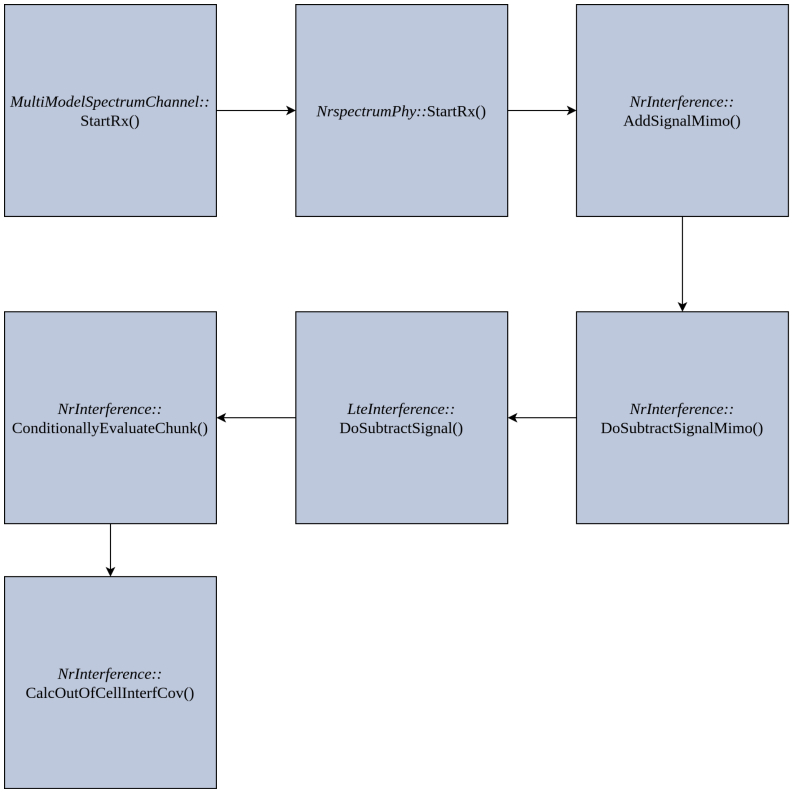

Fig. 2 - Simulation run until we got an abort.

To bypass this problem, my mentors and I proposed two approaches, which are discussed below.

Proposed solutions:

To achieve support for non-su-mimo channel models, we need to provide a bypass or even a new implementation in (maybe nr-spectrum-phy.cc) an NR class to deal with these models. The first implementation could be to go back to the 2.6 implementation, which does not need the spectrum channel matrix to evaluate the interference signals from all out-of-cell interferers[https://gitlab.com/cttc-lena/nr/-/blob/5g-lena-v3.0.y/doc/source/nr-module.rst?ref_type=heads]. We can achieve this by making a condition to check if the spectrum class assigns a spectrum channel matrix to the *SignalSpectrumParameters* (SSP), as shown in Fig. 3, which stands as a variable that has the signal parameters.

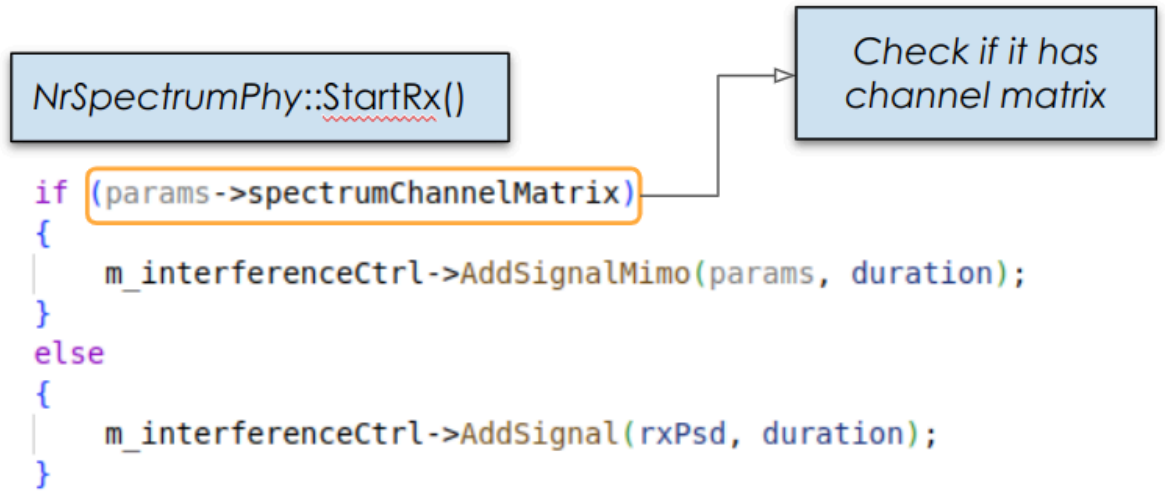

Fig. 3 - If condition to verify if the SSP contains a spectrum channel matrix

Using this approach, we have a simple (re)implementation because the classes have already been deployed (thanks to the nr/ns-3 team). We just need to assign the method using the condition. Unfortunately, this implementation is limited because all the upgrades deployed in the 3.0 version (and others being deployed) will not fit these channel models.

Another investigated approach creates a spectrum channel matrix using the current PSD, similar to how the 3GPP class does, as shown in Fig. 4.

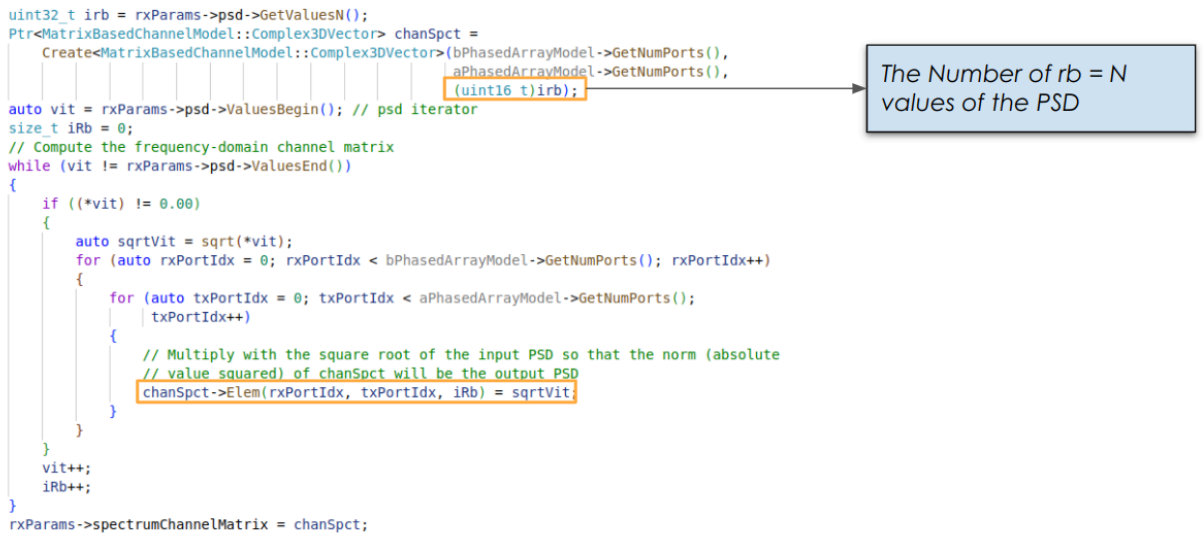

Fig. 4 - Create a (spectrum) channel matrix using the calculated received PSD.

This implementation works, and the equation is the same as in 3GPP. The problem is that we are trying to fit it into the NR module because we need it, not because the spectrum class defines the channel matrix for a given channel model. Therefore, we can compare both approaches in Table 1-2.

Advantages	Disadvantages
We know it works (used in 2.6)	Not compute interference signals from all out-of-cell interferers
Easy reimplementation	Maybe not upgradeable
specific to two-ray: it is already calibrated	

Table 1: Advantages and disadvantages of using the first approach.

Advantages	Disadvantages
Upgradable	I don't know if this is right; maybe it needs a more profound study to fit it into other channel models
Easy implementation	(maybe) Difficult validation
Compute interference signals from all out-of-cell interferers	

Table 2: Advantages and disadvantages of using the second approach.

We still need a new approach based on all these approaches and pointing out the most important advantages and disadvantages. To check both approaches in code, please see [1] [2]. One thing to consider is that we are dealing with different channel models, each with specific features, and we have to ensure these features are supported. For example, NYUSIM, FTR, and 3GPP are *PhasedArraySpectrumPropagationLossModel,* so they have antenna features. If you have a spectrum model (which is not a phased array one), you may have trouble importing into 5G-Lena. In the following weeks, I will propose a tutorial to help the user import his/her channel model in 5G-Lena/ns-3.

Solution and conclusion:

- After some discussion, we decided to upgrade the second approach. Instead of returning to the 2.6 version, we can create the (spectrum) channel matrix in the nr-spectrum-phy class, which may check if the signal has a channel matrix. If it does not, we calculate it using a similar method in the 3GPP class.

Milestone 2: Interface implementation for selecting the channel model via the NR helper class

Problem Overview:

As mentioned in the introduction of the project milestones, the band configuration and initialization are aligned with the 3GPP object creations (spectrum, propagation, and channel condition). Because of this, any other channel model is not supported without changing these methods of initialization and configuration.

NrHelper and *CcBwpCreator* are the classes responsible for initialization and configuration, respectively. As described in the *Week 1 - overview* section, *CcBwpCreator* only allows you to create BWPts with different scenarios (e.g., RMa, UMa). Also, the scenarios have the channel condition (if LOS or NLOS) merged in one enum (e.g., RMa_Los, RMa_NLos). This information is typically separated.

Because the user can only configure the type of scenario using the standard method, the initialization happens in *InitializationOperationBand* only with 3GPP configuration. Therefore, the spectrum, propagation, and channel conditions are set only with 3GPP, where the scenario defines which condition, spectrum, and propagation class will be used.

Proposed Solution:

To achieve a more modular approach, the user can select different channel models, scenarios, and conditions. To achieve this, I created two other enums in the struct that hold the band information (*BandWidthPartInfo)*: Channel and Condition. Also, as Gabriel recommended, I changed from a simple enum to an enum class, as shown in Fig. 5.

```
/**
 * \brief Different types for th
 */
enum class Scenario
{
    RMa,              //!< RMa
    UMa,              //!< UMa
    UMi_StreetCanyon, //!< UMi_S
    InH_OfficeOpen,   //!< InH_O
```

```
/**
 * \brief Different types
 */
enum class Channel
{
    ThreeGpp, //!< 3GPP
    TwoRay,   //!< TwoRay
    NYU       //!< NYU
} m_channelModel{Channel:
```

```
enum class Condition
{
    LOS,  //!< Always l
    NLOS, //!< Never Li
    Mixed //!< Mixed

} m_condition{Conditior
```

Fig 5.1 - Scenario enum Fig. 5.2 - Channel enum Fig. 5.3 - Condition enum

Fig. 5 - Different enum classes that the user can change.

Increasing the number of enum classes that define the BWP in terms of channel condition, scenario, and channel model makes it more modular. The user can mix all the possible attributes, and we handle the wrong combinations. The wrong combinations will break the entire simulation, and we do this using the already implemented *abort_msg* in spectrum classes, e.g., in *TwoRaySpectrumPropagationLossModel:*

```
  if (scenario != "RMa" && scenario != "UMa" && scenario != "UMi-StreetCanyon" &&
scenario != "InH-OfficeOpen" && scenario != "InH-OfficeMixed" && scenario != "V2V-Urban" &&
scenario != "V2V-Highway")
{
NS_ABORT_MSG("Unknown scenario (" + scenario +"), “choose between RMa, UMa, UMi-StreetCanyon, InH-OfficeOpen, InH-OfficeMixed, V2V-Urban or V2V-Highway");
}
```

After creating the enum classes in the *BandWidthPartInfo,* we need to ensure the user can access them to install them on all band component carriers. To achieve this, we need to update the *SimpleOperationBandConf* struct:

```
        SimpleOperationBandConf(
            double centralFreq = 28e9,
            double channelBw = 400e6,
            uint8_t numCc = 1,
                                BandwidthPartInfo::Scenario scenario =
BandwidthPartInfo::Scenario::RMa,
                               BandwidthPartInfo::Condition condition =
BandwidthPartInfo::Condition::Mixed,
                              BandwidthPartInfo::Channel channelModel =
BandwidthPartInfo::Channel::ThreeGpp)
            : m_centralFrequency(centralFreq),
              m_channelBandwidth(channelBw),
              m_numCc(numCc),
              m_scenario(scenario),
```

```
            m_condition(condition),
            m_channelModel(channelModel)
```

Now, not only the m_scenario variable is being set in the band, but also the channel condition variable (m_condition) and the channel model (m_channelModel), which are being set with default values (scenario = RMa, Condition = Mixed and Channel = ThreeGpp). With the band already configured, we can create it using the *CreateOperationBandContiguousCc*. Because we are holding the information for each BWP of all of the component carriers, we have to update the creation of the BWP's component carrier:

```
    // Create a CC with a single BWP
                    std::unique_ptr<ComponentCarrierInfo>       cc(new
ComponentCarrierInfo());
    InitializeCc(cc, ccBandwidth, lowerFreq, ccPosition, ccId);

    double bwpBandwidth = ccBandwidth / bwpNumber;

    for (uint8_t i = 0; i < bwpNumber; ++i)
    {
        std::unique_ptr<BandwidthPartInfo> bwp(new BandwidthPartInfo());
            InitializeBwp(bwp, bwpBandwidth, cc->m_lowerFrequency, i,
m_bandwidthPartCounter++);
        bwp->m_scenario = scenario;
        bwp->m_channelModel = channelModel;
        bwp->m_condition = condition;
        bool ret = cc->AddBwp(std::move(bwp));
        NS_ASSERT(ret);
    }
```

As you can see, the BWP of the current component carrier contains not only the scenario but the channel condition and the channel model information, which can now be initialized. Therefore, now in the *NrHelper* - *InitializeOperationBand*, which will be responsible for:

- Setting the type ID of the object factory variables
- Create the propagation, channel condition, and channel model objects
- Define the user's desired attributes to the channel/propagation or channel condition classes.

Following these tasks, we will use the same type ID initialization approach; see *Week 1—Overview,* with some modifications. We initialized the propagation and channel condition type ID using the scenario. Instead of using only the scenario, we will use the band's channel model, as shown in Fig. 6.

```
static std::map<std::pair<BandwidthPartInfo::Scenario, BandwidthPartInfo::ChannelModel>,
                InitPathLossFn>
    initLookupTable{
        {BandwidthPartInfo::RMa,
        {{BandwidthPartInfo::RMa, BandwidthPartInfo::ThreeGpp},
         std::bind(&InitRma, std::placeholders::_1, std::placeholders::_2)},
        {BandwidthPartInfo::RMa_LoS,
        {{BandwidthPartInfo::RMa_LoS, BandwidthPartInfo::ThreeGpp},
         std::bind(&InitRma_LoS, std::placeholders::_1, std::placeholders::_2)},
        {BandwidthPartInfo::RMa_nLoS,
        {{BandwidthPartInfo::RMa_nLoS, BandwidthPartInfo::ThreeGpp},
```

Fig. 6 - A new map was used to configure the Type ID.

The current BWP scenario and channel model will set my channel condition and propagation object factories. As shown below, the spectrum object factory will be set using the BWP channel model information as a map value containing all the possible channel models (NYUSIM, 3GPP, and FTR).

```
   m_channelTypeIdlMap[BandwidthPartInfo::ThreeGpp] = {
        ThreeGppSpectrumPropagationLossModel::GetTypeId()};
    m_channelTypeIdlMap[BandwidthPartInfo::TwoRay] = {
       TwoRaySpectrumPropagationLossModel::GetTypeId()};

#ifdef ENABLE_NYU
m_channelTypeIdlMap[BandwidthPartInfo::NYU]                                    =
{NYUSpectrumPropagationLossModel::GetTypeId()};
#endif // ENABLE_NYU
```

Because we set the correct type ID for the three conditions, spectrum, and propagation, we can create our objects:

```
auto channelConditionModel =

m_channelConditionModelFactory.Create<ChannelConditionModel>();
auto plm = m_pathlossModelFactory.Create<PropagationLossModel>();
auto channel =
```

```
m_spectrumPropagationFactory.Create<PhasedArraySpectrumPropagationLossModel>();
```

And set the right attributes to it:

```
                                        plm->SetAttribute("Frequency", DoubleValue(bwp->m_centralFrequency));
                                plm->SetAttribute("ChannelConditionModel", PointerValue(channelConditionModel));
                channelObject->SetAttribute("ChannelConditionModel",

PointerValue(channelConditionModel));
                                    channelObject->SetAttribute("Frequency", DoubleValue(bwp->m_centralFrequency));
                                     channelObject->SetAttribute("Scenario", StringValue(bwp->GetScenario()));
```

NOTE: Channel object is a variable that can be either a matrix-based channel model class or a PhasedArraySpectrumPropagationLoss. If it is matrix-based, we configure our values (frequency, scenario, and channel condition) directly to the matrix-based class, or if it is not matrix-based, we configure it directly to the spectrum class. We do this by doing the following:

```
                Ptr<Object> channelObject =
                    hasChannelMatrixModel
                                                                                ? DynamicCast<Object>(matrixChannelClassPtr.Get<MatrixBasedChannelModel>())
                        : DynamicCast<Object>(channel);
```

Solution and conclusion:

- Many functions are used to set the TypeId (e.g., Init3GPPUma); all are similar and could be removed. Instead, all the initialization would happen inside the main function based on the TypeIds.
- Check whether the channel model can be aggregated to PhasedArraySpectrumPropagationLossModel to obtain it with GetObject; note: see how MobilityModel is aggregated to Node and obtained; read in ns-3 documentation about object aggregation and usage.

- Explain configuring channel, pathloss, and condition using the band ID (not explained in this section; please see *week 4*)
- Create initial draft MR/MRs

Presentation:

https://docs.google.com/presentation/d/1CI2sj_lnZ7TCp5Iux2spw3ibnm_O5rw-HCh7NeU5JcA/edit?usp=sharing

Week 4:

Milestone 2: Interface implementation for selecting the channel model via the NR helper class [continue]

After some discussion and conclusions, we decided that there are better approaches than having a big map with too many *typedef* functions. Instead, I proposed creating a new function in the *CcBwpCreator*, which will be responsible for getting the band ID information (propagation, spectrum, and channel condition):

```
std::tuple<std::string, std::string, std::string>
BandwidthPartInfo::GetBandTypeIdInfo() const
{
                    return       std::make_tuple(GetPropagationTypeId(),
GetChannelModelTypeId(), GetConditionTypeId());
}
```

The idea of this function is to return a tuple that contains the type ID information, where the values are the propagation, channel, and condition type IDs. Each *Get${information}TypeId* returns a string that contains the current user-selected scenario/condition/channel.

Propagation Type ID:

```
std::string
BandwidthPartInfo::GetPropagationTypeId() const
{
```

```
    NS_LOG_FUNCTION(this);
    auto currentModel = m_channelModel == Channel::TwoRay ? "ThreeGpp" :
GetChannelModel();
      if (m_scenario == Scenario::InH_OfficeMixed || m_scenario ==
Scenario::InH_OfficeOpen)
    {
                                return "ns3::" + currentModel +
"IndoorOfficePropagationLossModel";
    }
    else
    {
         // Use the same propagation/channel condition model for both the
3GPP and TwoRay
                  return "ns3::" + currentModel + GetScenario() +
"PropagationLossModel";
    }
}
```

NOTE: FTR uses the same propagation and type ID of 3GPP. That's why we need to do:

```
auto currentModel = m_channelModel == Channel::TwoRay ? "ThreeGpp" :
GetChannelModel();
```

As you can see, if the user selects the InH_*, the Indoor propagation model is returned ("ns3::3GPPIndoorOfficePropagationLossModel"), but if the user chooses another scenario, the propagation could be either *"ns3::ThreeGppRmaPropagationLossModel"* or *"ns3::NYURmaPropagationLossModel"*. The *GetScenario()*, *GetChannelModel()* and *GetCondition()* (we will use it after) returns the translation from enum to string of the user-selected feature (e.g., Channel:TwoRay → "TwoRay").

Channel (spectrum) Type ID:

```
std::string
BandwidthPartInfo::GetChannelModelTypeId() const
{
    NS_LOG_FUNCTION(this);
    return "ns3::" + GetChannelModel() + "SpectrumPropagationLossModel";
}
```

This one has simpler logic because the type ID of the spectrum class is typically defined by `"ns3::" + GetChannelModel() + "SpectrumPropagationLossModel"` (e.g., “ns3::TwoRaySpectrumPropagationLossModel”).

Condition Type ID:

```
std::string
BandwidthPartInfo::GetConditionTypeId() const
{
   NS_LOG_FUNCTION(this);

       if (m_scenario == Scenario::UMa_Buildings || m_scenario == Scenario::UMi_Buildings)
   {
       return "ns3::BuildingsChannelConditionModel";
   }

   if (GetCondition().empty())
   {
            auto currentModel = m_channelModel == Channel::TwoRay ? "ThreeGpp" : GetChannelModel();
                return "ns3::" + currentModel + GetScenario() + "ChannelConditionModel";
   }
   else
   {
       return "ns3::" + GetCondition() + "ChannelConditionModel";
   }
}
```

To get the condition type ID, we need to verify first if the user is choosing a scenario with buildings condition (`m_scenario == Scenario::UMa_Buildings || m_scenario == Scenario::UMi_Buildings`); in that case, we need to ensure that the conditions type ID is the buildings one. If the user did not choose it, we need to verify what type of conditions he selected using the *GetCondition()*. The user can choose three conditions: LOS, NLOS, and Mixed. If he/she selects mixed, the *GetCondition()* function returns an empty string, and the current function will return `"ns3::" + currentModel + GetScenario() + "ChannelConditionModel";` (e.g., “ns3::NYURmaChannelConditionModel). But if the user selects LOS (“AlwaysLos”) or NLOS (“NeverLos”), the type id will be `"ns3::" +`

`GetCondition()` + `"ChannelConditionModel"` (e.g., "ns3::AlwaysLosChannelConditionModel).

Now that we set all the user-selected type IDs in a tuple, we must call it within the nr-helper InitializeOperationBand. However, we need to ensure two things firstly:

- if the user is selecting the correct combination
- if the user is trying to use NYU without it being installed

To check if the user is trying to use NYU without it being installed, we used a conditional build, which will be defined in our next weeks:

```
#ifndef ENABLE_NYU
                                NS_ABORT_MSG_IF(bwp->m_channelModel   ==
BandwidthPartInfo::Channel::NYU,
                                 "Install the NYU module to use the NYU
channel model.");
#endif // ENABLE_NYU
```

To check if the combination is correct, we first defined a giant lookup table with all the possible combinations, which a part will be shown below:

```
// Supported combinations of channel model, channel condition, and
scenario
std::set<std::tuple<BandwidthPartInfo::Channel,
                    BandwidthPartInfo::Condition,
                    BandwidthPartInfo::Scenario>>
   m_supportedCombinations = {
       // NYU-RMA
       {BandwidthPartInfo::Channel::NYU,
        BandwidthPartInfo::Condition::Mixed,
        BandwidthPartInfo::Scenario::RMa},
```

After we verify:

```
            if (m_supportedCombinations.find(
                        std::make_tuple(currentChannel, bwp->m_condition,
bwp->m_scenario)) ==
                m_supportedCombinations.end())
            {
```

```
                NS_ABORT_MSG(
                      "Unsupported combination of channel model, channel
condition and scenario");
            }
```

If the combination is incorrect (does not exist), the entire simulation will be aborted. But if it exists, then we do:

```
             // Get the band info that contains the propagation, channel
model, and channel condition
            auto bandInfo = bwp->GetBandTypeIdInfo();

            // Set the type ID of the factories
            m_pathlossModelFactory.SetTypeId(std::get<0>(bandInfo));

m_spectrumPropagationFactory.SetTypeId(std::get<1>(bandInfo));

m_channelConditionModelFactory.SetTypeId(std::get<2>(bandInfo));
```

Solution and conclusion:

The channel condition enum should have one more value called "Buildings." The mapping function should also call another mapping function that maps each enum into corresponding TypeIds. I need to check how attributes of the enum type work in ns-3 and how they can be implemented in our context, and I need to add the Two-ray channel model to the supported combinations.

- **First task**: Refactor BandwidthPartInfo to allow the user to configure it with three independent strings instead of enums:
    - BandwidthPartInfo needs to be a class
    - Pass these three string arguments directly to attributes of bwp that are of enum type, so at least one conversion is done "automatically"
    - Move *Buildings* to the channel condition enum class
    - Add the Two-ray channel model to the supported combinations
- **Second task**: Translation from enum to TypeId
    - Check if the user-selected combination works (I think it is better to call it here before we make a conversion from enum to TypeIDs, but I can

also do this later. I just have to ensure that the user selected a correct combination)

- Convert from enum to TypeID using the functions I already created (instead of returning a string, return the right TypeId)
- Improve the *GetConditionTypeId()* function and define it to set only the LOS, NLOS, and Buildings.