Faculdade de Engenharia da Universidade do Porto

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

A link level simulation framework for Machine Type Communication towards 5G

Rui Nunes

V.1.1

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major Telecomunicações

Orientador: Professor Ricardo Morla

14 de Julho de 2017

Abstract

5G is currently being standardized, and many aspects of the physical layer are yet to be decided, particularly related to Machine Type Communication (MTC). Building upon the 3GPP release 13 specifications for MTC, this thesis presents a link level simulation framework, allowing for quick prototyping of ideas and concepts towards 5G.

Throughout this work, the architecture and main building blocks of the simulation framework are described, and implementation and configuration concepts are presented. The support for both standard and non-standard 3GPP configurations is illustrated, including OFDM numerology, pilot configuration and channel masking options. Simulation results are then presented, specifically targeting coverage enhancements topics for MTC. Different combining strategies are explored, followed by an investigation showing the importance of minimizing overhead for MTC applications.

Acknowledgments

I would like to thank all those who made this work possible. At FEUP, Professor Ricardo Morla for accepting to supervise this thesis, and for the invaluable support and guidance. At Sony Mobile, Hares Mawlayi and Peter Karlsson for the opportunity, Basuki Priyanto for the expertise, insight, time and patience, Göran Ekberg and Abbas Sandouk for doing magic in scheduling.

Finally, a special thanks to Marta Jani, for the constant support and motivation during this period.

Contents

List of figures

List of tables

Abbreviations and Acronyms

3G	3rd Generation
3GPP	Third Generation Partnership Project
4G	4th Generation
5G	5th Generation
ARQ	Automatic Repeat Request
AWGN	Additive White Gaussian Noise
BCH	Broadcast Channel
BER	Bit Error Rate
BLER	Block Error Rate
CAT-0	3GPP category 0 device
CAT-M1	3GPP Category M1 device
CB	Code Block
CC	Chase Combining
CFO	Carrier Frequency Offset
CRC	Cyclic Redundancy Check
CRS	Cell-specific Reference Signals
eMTC	enhanced Machine-Type Communication
EPA	Extended Pedestrian A model
EVA	Extended Vehicular A model
ETU	Extended Typical Urban model
FDD	Frequency Division Duplex
FFT	Fast Fourier Transform
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
HARQ	Hybrid Automatic Repeat Request
HSDPA	High-Speed Downlink Packet Access
HSUPA	High-Speed Uplink Packet Access
IR	Incremental Redundancy
LLR	Log-Likelihood Ratio
LTE	Long Term Evolution
MIB	Master Information Block
MRC	Maximum-Ratio Combining
MTC	Machine-Type Communication
NR	New Radio
OFDM	Orthogonal Frequency-Division Multiplexing

OFDMA	Orthogonal Frequency-Division Multiple Access
PCFICH	Physical Control Format Indicator Channel
PHICH	Physical Hybrid-ARQ Indicator Chanel
PDCCH	Physical Downlink Control Channel
PDSCH	Physical Downlink Shared Channel
PRB	Physical Resource Block
PSS	Primary Synchronization Channel
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase-Shift Keying
RB	Resource Block
RBn	Resource Block number
RV	Redundancy Version
RX	Receiver
SFN	System Frame Number
SNR	Signal-to-Noise Ration
SSS	Secondary Synchronization Channel
TB	Transport Block
TDD	Time Division Duplex
TTI	Transmission Time Interval
TX	Transmitter
UE	User Equipment
UMTS	Universal Mobile Telecommunications System

Chapter 1

Introduction

This chapter introduces the scope of the dissertation, presenting the assumptions and motivations for this work. The structure of the thesis is detailed at the end of the chapter.

1.1 - Overview of the physical layer aspects and MTC

1.1.1. An overview of the Physical Layer evolution towards 5G

Even though markedly designed for voice services, the early second generation GSM standards included already support for transparent, and non-transparent data services over the circuit-switched network, with a Radio Link Protocol supporting reliable communication over the air interface [20]. The introduction of General Packet Radio Service (GPRS), in 2000, brought the packet based concept to the second generation cellular network, backed by a new radio protocol stack and a new packet switched core network [18]. However, relying on the same GSM physical layer technology, optimized for circuit switched voice services, GPRS was a compromise between backwards compatibility and service flexibility.

The third generation, UMTS, started deployment end of 2003 building upon the same GPRS core network infrastructure, but with a radically new air interface approach. The physical layer architecture was designed to be flexible, to accommodate a number of different services, at different quality of services, and for data rates up to 384 Kbit/s. The core of this flexibility lies on the new transport channel processing architecture introduced in the physical layer [21]. It became possible, not only to design services with different channel coding algorithms, different coding rates and different transmission time intervals, but these services could also be multiplexed and used simultaneously.

The UMTS physical layer flexibility was further improved with the introduction of HSDPA in 2006 and later HSUPA. While the initial releases of UMTS relied on a relatively slow higher layer based, and mostly static block size allocation, HSDPA introduced new physical layer signaling channels, allowing for much faster allocation times. Transport block sizes and modulation could now be changed dynamically, effectively matching the link to the channel conditions at much higher rates than before. Additionally, a new retransmission scheme, HARQ, was introduced, providing fast retransmission times, and improved decoding with soft combining of all retransmissions, at the receiver [22].

The physical layer of the 4th generation, Long Term Evolution (LTE), built largely on the concepts op HSDPA. While the access technology changed from CDMA to OFDMA, the transport channel processing architecture, inherited most of the concepts from the previous generation, even though enhanced by the extra flexibility provided by OFDMA [1].

The physical layer of the 5th generation New Radio (NR) is expected to be largely based on the current LTE generation’s architecture, possibly more than in any previous iteration. And while there are important changes, namely new algorithms for channel coding, new antenna related technologies and larger bandwidth, scalable transmission times and OFDM numerologies are expected to be key components in allowing the support for the three target 5G use cases: enhanced mobile broadband, ultra-high reliability & low latency and Massive IoT [17][24].

1.1.2. Machine Type Communication within LTE

Wireless connectivity of devices, without human intervention is referred within 3GPP as Machine Type Communication (MTC). As we have seen, data services have been possible throughout the different generations of cellular networks, enabling already many possible MTC application. However, the number of connected devices over cellular networks is expected to grow massively over the next few years [58][27], especially within the class of devices including sensors and actuators, typically associated with low cost and long battery life, often located in places with poor cellular coverage.

In order to accommodate these devices, 3GPP have standardized new User Equipment (UE) categories for LTE, namely category-0 (CAT-0), in release 12, and category-M1 (CAT-M1) in release 13. CAT-M1 devices are expected to support a coverage gain of up to 21 dB relative to legacy LTE devices [41], and at a fraction of the cost, due to relaxed requirements on bandwidth, transmit power, and antenna configuration [3].

1.2 - An overview of related simulation environments in Matlab

A number of LTE link-level simulators have been developed over the years, both commercial and open-source.

In [61], a commercial LTE MATLAB based link level simulator is available, supporting physical layer implementation of up to release 10 of the 3GPP specifications.

In [60], LTE link-level simulators for both uplink and downlink have been developed in MATLAB, with a free license for academic and non-commercial use. It contains a vast amount of features, including several multi-antenna configurations, a variety of fading profiles, and channel estimation and equalization methods. Even though no release compliance is clearly stated, the user manual suggests a pre-release 12 3GPP compliance due to lack of 256QAM support [59].

Several additional open source simulations and frameworks have been presented over the years, as [63] or [64], however, none has been updated to 3GPP release 13.

MATLAB provides an LTE specific toolbox under the name of LTE System Toolbox [62]. It contains uplink and downlink processing chains and all standardized physical channels and signals, as well as all multi-antenna transmission schemes up to release 12 of the 3GPP specifications. The toolbox has recently been updated to include 5G channel models as per [23]. While including a rich set of tools, this cannot be seen as a finished link-level simulator of framework.

Even though there are a number of available simulation environments, to the best of our knowledge, there is no available release 13 3GPP compliant solution. Also, within the most updated environments, as in [60], the design goal was clearly to implement LTE specific features, timing and OFDM numerologies, so that the flexibility to prototype non-standard solutions, including different OFDM numerologies, was not an intention [59]. By presenting a framework able to implement the MTC relevant 3GPP release 13 features, and by including an open frame structure and OFDM numerology, this work is expected to be a valuable tool in prototyping new ideas towards 5G.

1.3 - Motivation and contents

The main contribution of this thesis is the development of a link-level simulation framework, capable of prototyping ideas and concepts for MTC devices, specifically in areas of coverage enhancement and power consumption. While having CAT-M1 devices, as a starting point, the aim is to develop a framework capable of adapting to the evolving 3GPP standardization towards 5G, by implementing a flexible timing and OFDM numerology, an open modular approach, as well as a fully configurable system.

This thesis was sponsored by the Research & Standardization Organization at Sony Mobile, in Lund, Sweden.

This thesis is organized as follows:

- Chapter 2 gives an overview of the framework requirements, architecture and main data structures.
- Chapter 3 describes the implementation details of key modules within the framework.
- Chapter 4 presents simulation results, illustrating the framework flexibility, and validating key components with published results.
- Chapter 5 concludes this thesis with a critical analysis on main contribution as well as suggestions for future improvements.

Chapter 2

The Simulation Framework overview

This chapter provides an overview of the simulation framework. It starts by describing the high level requirements and basic building blocks, in section 2.1- , followed by an overview of the main data structures used, in section 2.2- .

2.1- Requirements, architecture and basic building blocks

The simulation framework was developed to prototype Machine Type Communication (MTC), with the LTE release 13 of the 3GPP specifications as baseline. The aim is to be able to accommodate future releases, including the next generation New Radio, currently being defined. The framework was written in MATLAB, reusing, as much as possible, the MATLAB system objects. The framework was developed with the following design goals:

- Fast prototyping of new ideas and concepts.
- Target MTC devices.
- Support LTE FDD PDSCH and devices with category M1, as a starting point.
- Modular, with clear interfaces for easy maintenance.
- Easy addition of new modules.
- Generic, to accommodate a growing number of algorithms.
- Reconfigurable, for simple operation from a main script, including disabling or bypassing functions.
- Easy comparison of different simulation configurations.
- Possibility to stop and resume simulations from the last simulation point.
- Possibility to store the simulation results and related configuration for further data analysis.
- Accessible through one single API.
- Written in MATLAB.

A high level architecture overview of the framework is presented in Figure 2.1. The framework provides one basic service. Having received a given configuration and a target SNR range, the framework runs the requested simulation, returning the resulting BER and BLER performance values for the each point in the provided SNR range. For the purposes of this framework, each SNR point within a configuration is called a sub-instance, while a particular configuration is called an instance.

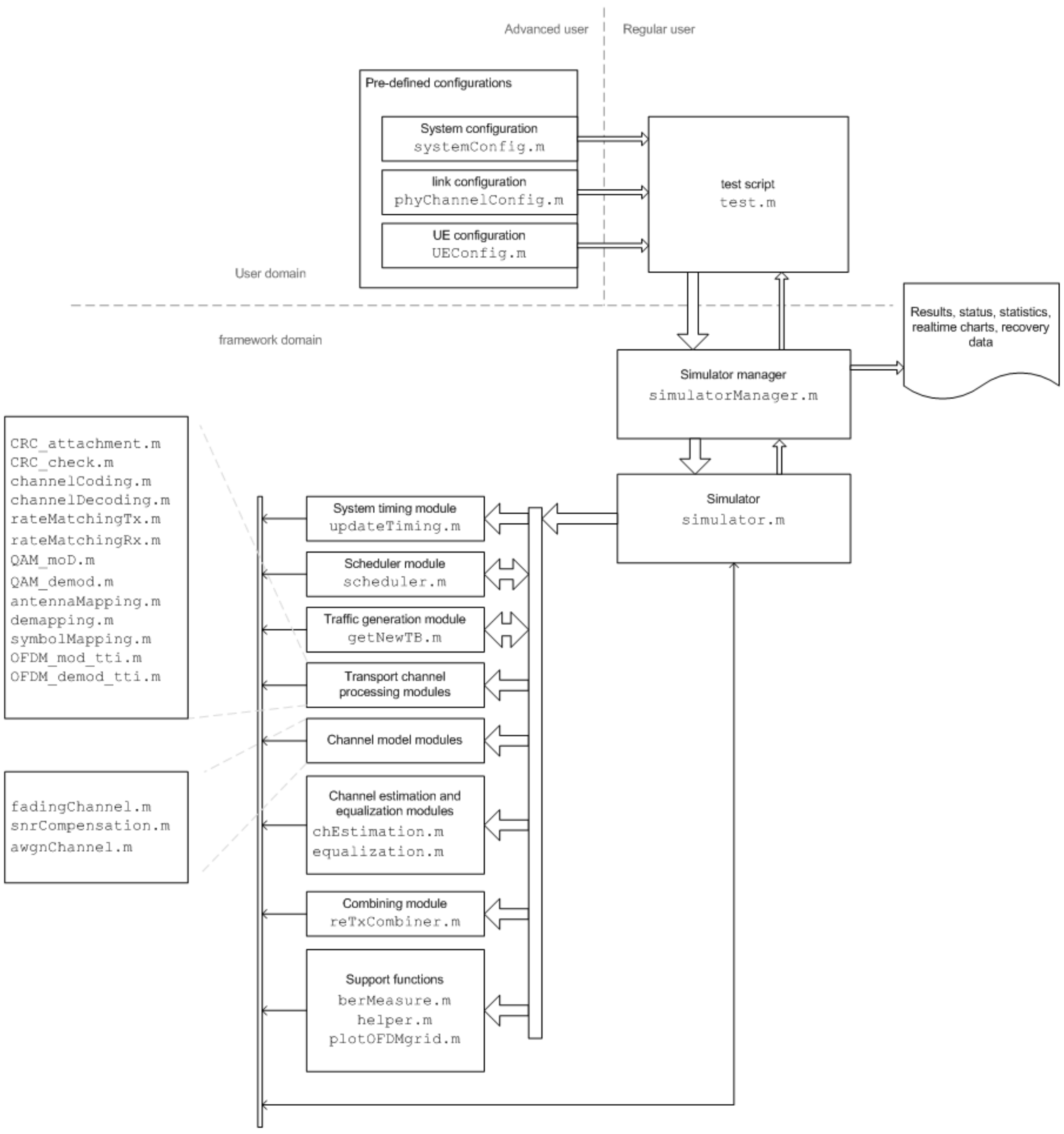


Figure 2.1 – Framework architecture overview

Typically, the framework will be requested to run sequentially through several configuration instances, each with several sub-instance.

Each configuration instance is defined by a new *config* data structure containing the entire desired framework parametrization, including a number of target SNR points to be simulated, the sub-instances.

Information exchange between the test script and the simulation framework is done through one single data structure containing a collection of all individual configuration instances to be simulated. This is the run data structure. These data structures are detailed in section 2.2-

There are three main entities within the framework, the test script, the simulation manager and the simulator. These will be described next.

2.1.1. The test script

The task of the test script is to populate the run data structure with all required configuration instances. This can be done with the support of predefined configurations. Having all configurations defined, the script then makes one single call to the framework by calling the *simulationManager* API with the run structure as argument.

Once the simulation on all configuration instances is over, the framework returns the simulation results for further post processing by the script.

An overview of a typical test script is shown in Figure 2.2, and an example script is provided in Annex A.5 - .

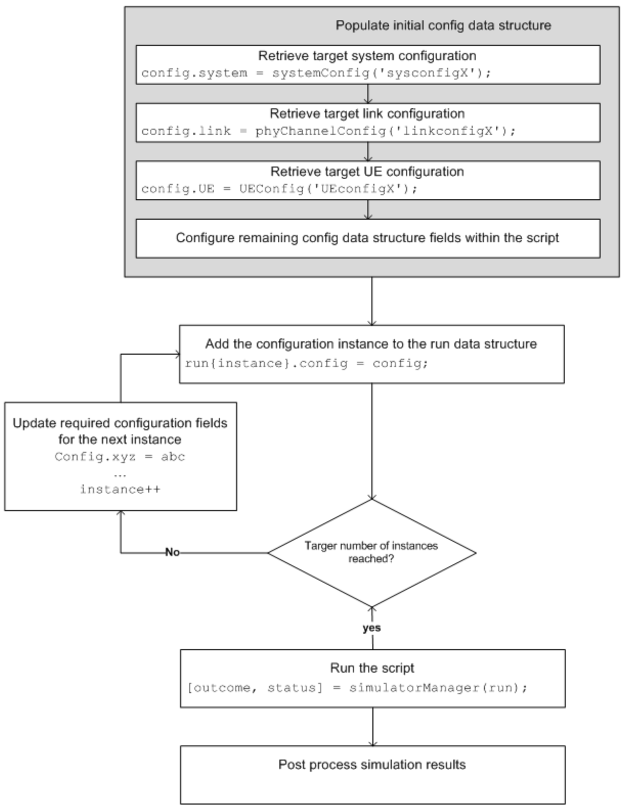

Populate initial config data structure
Retrieve target system configuration
config.system = systemConfig('sysconfigX');
Retrieve target link configuration
config.link = phyChannelConfig('linkconfigX');
Retrieve target UE configuration
config.UE = UEConfig('UEconfigX');
Configure remaining config data structure fields within the script
Add the configuration instance to the run data structure
run{instance}.config = config;
Update required configuration fields for the next instance
Config.xyz = abc
...
instance++
No
Targer number of instances reached?
yes
Run the script
[outcome, status] = simulatorManager(run);
Post process simulation results

Figure 2.2 – Typical test script overview

2.1.2. The simulation manager

The simulation manager handles the interface towards the test script and simulator. Having been invoked by a test script, the simulation manager will sequentially step through all configuration instances, calling the simulator for each particular sub-instance. This allows immediate access to every finished simulated SNR point.

Additionally the simulation manager stores the run data structure, together with the latest results from the simulator in the file system. This has two purposes, allow access to the very latest simulated results, on a sub-instance level, and allow for the simulation to be terminated and resumed from the last sub-instance. The processing flow overview of the simulation manager is shown in Figure 2.3.

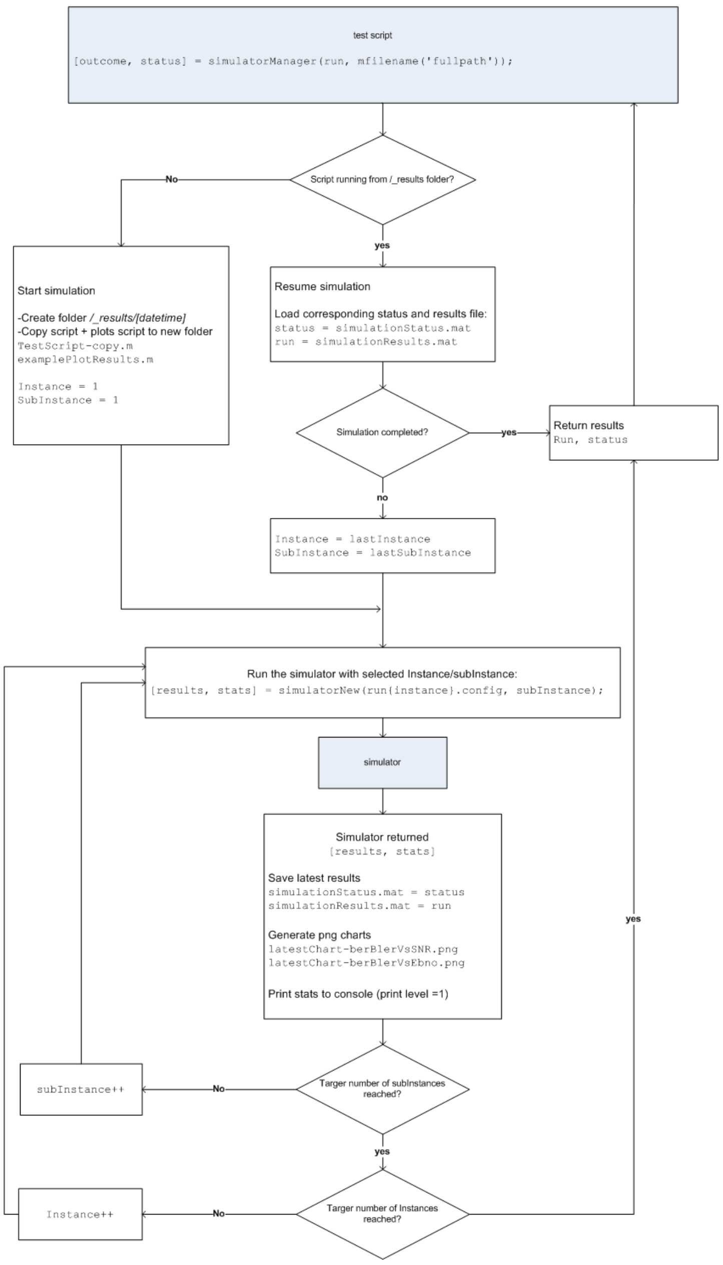

test script
[outcome, status] = simulatorManager(run, mfilename('fullpath'));
Script running from /_results folder?
No
yes
Start simulation
-Create folder /_results/[datetime]
-Copy script + plots script to new folder
TestScript-copy.m
examplePlotResults.m
Instance = 1
SubInstance = 1
Resume simulation
Load corresponding status and results file:
status = simulationStatus.mat
run = simulationResults.mat
Simulation completed?
yes
no
Return results
Run, status
Instance = lastInstance
SubInstance = lastSubInstance
Run the simulator with selected Instance/subInstance:
[results, stats] = simulatorNew(run(instance).config, subInstance);
simulator
Simulator returned
[results, stats]
Save latest results
simulationStatus.mat = status
simulationResults.mat = run
Generate png charts
latestChart-berBlerVsSNR.png
latestChart-berBlerVsEbno.png
Print stats to console (print level =1)
yes
Targer number of subInstances reached?
No
subInstance++
yes
Targer number of Instances reached?
No
Instance++

Figure 2.3 – Simulator manager overview

2.1.3. The simulator

The *simulator.m* module is the core of the framework. Detailed description of the different modules used by the simulator will be provided in Chapter 3. The processing flow of the simulator is shown in Figure 2.4.

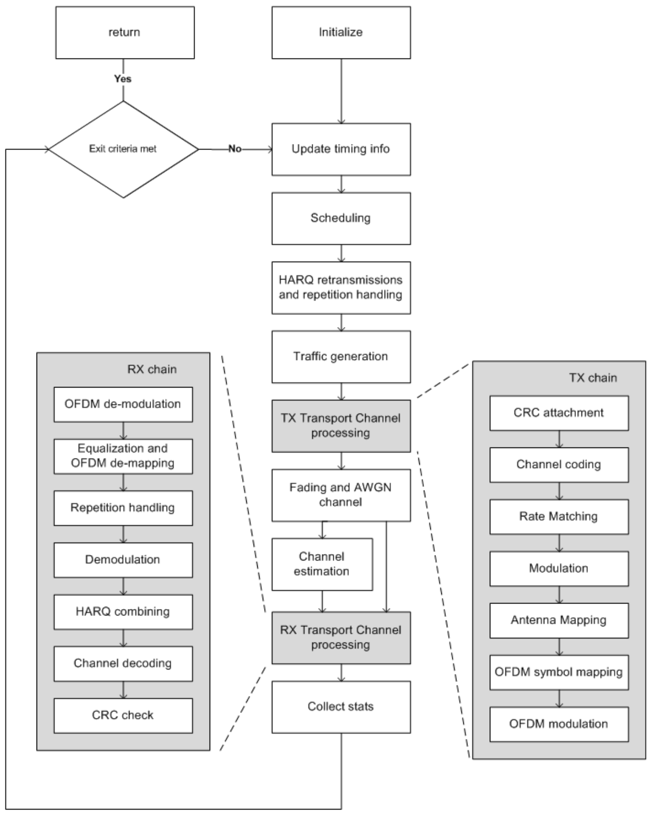


Figure 2.4 – Simulator processing overview

The simulator will receive a configuration instance and sub-instance from the simulator manager. The simulator will loop through its processing chain until one of the defined exit criteria is met. The exit criteria are part of the configuration defined by the script, and may be based on target amount of traffic or errors. Having reached one of the exit criteria, the simulator returns the requested simulation results, as well as additional statistics.

2.2- Data structures and the pre-defined configuration modules

The details of the framework data structures are described next. Focus will be on the most relevant members. The complete list is provided in Annex A.4 - .

2.2.1. *The config data structure*

The fields in the *config* data structure are organized in a logical arrangement. In order to minimize the number of available configuration options to a regular user, the *config* structure is further subdivided into three additional logical data structures, the *system*, the *link* and the *UE* data structures. These structures are available through predefined modules as shown in Figure 2.5.

Additionally, the *config* data structure contains a number fields for high level configurations. These are to be initialized by the test script and define the general simulation environment, including the modules to be enabled, the SNR range, and some specific algorithm selections.

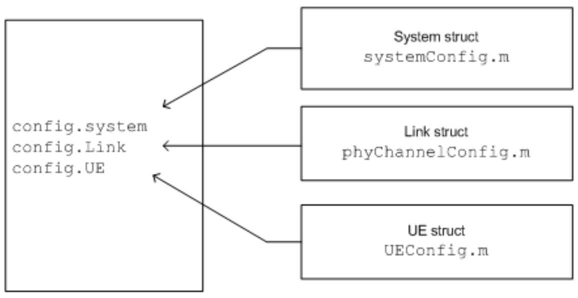


Figure 2.5 – Configuration data structures

The *system*, *link* and *UE* part of the *config* data structure are defined in the following subsections. The remaining *config* fields will be covered in the relevant modules. A full list of *config* fields is provided in Annex A.4 - , and a script example in Annex A.5 - .

2.2.2. *The system data structure*

The *system* data structure is responsible for the entire lower level physical channel configuration, including OFDM frame structure in time and frequency domain, cyclic-prefix configuration, pilot structure, channel masking, resource block configuration, FFT size and bandwidth configuration. The system configuration is not limited in any way to the LTE standardized configuration. This is the key to making the framework flexible to accommodate the new 5G OFDM configurations.

The system structure can be configured by calling the *systemConfig* function in the *systemConfig.m* module, pointing to a predefined configuration. The contents of the system structure are explained next.

The Resource Block definition (RB) is the basic building block of the system configuration. The RB is a structure containing two fields, the number of subcarriers and the number of OFDM symbols, as shown in Table 2.1. This defines the minimum OFDM resource allocation size in the frequency and time domain. The entire system bandwidth, timing and scheduling of the system will be reference to this RB definition. Time measures will be based on units of RB duration, defined by *system.RB.nsymbol*, and frequency measures will be given in given in multiples of *system.RB.nSC*.

For FDD LTE, *system.RB.nsymbol* defines one slot duration and is defined by either 6 or 7 OFDM symbols, depending on the usage of extended or normal cyclic prefix, respectively. In LTE, system.RB.nSC is fixed to 12 subcarriers.

Table 2.1 - Resource Block definition

Field	Description	LTE example
system.RB.nSC	Number of subcarriers per resource block	12
system.RB.nsymbol	Number of OFDM symbols per resource block	7*

* Normal cyclic prefix case. This defines the RB period as having 7 OFDM symbols, or 0.5 ms, a slot duration in LTE.

The subcarrier spacing is an important OFDM design parameter, directly impacting the performance of an OFDM system on particular channel conditions. It implicitly defines the OFDM symbol duration. This is defined with the field *System.SCspacing* as shown in Table 2.2. For LTE, subcarrier spacing is 15 KHz.

Table 2.2 - OFDM subcarrier spacing

Field	Description	LTE example
system.SCspacing	Subcarrier spacing in Hz	15000

The system bandwidth and FFT size are defined next. There are two bandwidth related fields, the transmission bandwidth and the channel bandwidth. The transmission bandwidth is the scheduled bandwidth and defined in number of Resource Blocks. The channel bandwidth is the total used system bandwidth. For LTE this information is given in [10]. The FFT size is related to the transmission bandwidth and needs to be higher than the maximum number of subcarriers. These fields are described in Table 2.3.

Table 2.3 - Bandwidth definition

Field	Description	LTE example
system.nRB	Available transmission bandwidth in number of RBs	50*
system.BW	Channel bandwidth in MHz	10
system.NFFT	FFT size for OFDM	1024*

* For 10 MHz bandwidth there are 600 subcarriers

The information defined so far in the system structure allows the calculation of additional information that is added to the system structure for easy reference by various framework modules. Calculation is done as shown in Table 2.4.

Table 2.4 - Sampling information

Field	Calculation	Description
system.TsOFDM	1/system.SCspacing	OFDM symbol duration
system.Ts	system.TsOFDM/system.NFFT	Sampling period
system.Fs	1/system.Ts	Sampling frequency

The Cyclic Prefix (CP) configuration is also referenced to the RB definition, namely with the number of OFDM symbols to be used. Each symbol in the RB definition will have its own CP definition, allowing full flexibility in order to have different CP durations on different OFDM symbols. The only assumption is that the pattern repeats with each new RB. The configuration is made with the field system.cpNSamples, defining an array with size *system.RB.nsymbol*, where each entry represents the number of samples used for the CP for each OFDM symbol as shown in Table 2.5.

Table 2.5 - Cyclic prefix definition

Field	**Description**	**LTE example**
system.cpNSamples[]	Number of samples per cyclic prefix for each OFDM symbol	[80 72 72 72 72 72 72]*

* LTE with Normal cyclic prefix and FFT size of 1024

The number of transmit antennas is defined in *system.nTx a*s shown in Table 2.6.

Table 2.6 - Number of downlink transmit antennas

Field	**Description**	**LTE example**
system.nTx	Number of TX antennas	2

The pilot configuration is defined with a new structure containing the pilot indexes for each antenna port, for the entire transmission bandwidth, and for one RB duration. This allows for a fully flexible pilot arrangements, including the LTE-like hexagon pattern, comb type, block type and others [30]. For LTE this defines the Cell Specific Reference Signals, CRS [11].

Table 2.7 - Pilot definition

Field	Description	LTE example
system.pilot.idx[]	pilot pattern for each antenna port	*
system.pilot.PowerBoost	Power boost for pilot symbols	1

* Example shown in the scheduler description

Having defined the basis of the OFDM configuration, it is possible to define additional channel masks occupying some of the OFDM resources. This is typically the case for physical layer signaling channels. Each channel may be configured with specific timing information including timing of first transmission as well as period of subsequent transmissions. This information will be used by the scheduler when building the OFDM grid and assigning user resources on a particular transmission time. The channels are defined as MATLAB cells with structures, where each channel is defined by a structure within a cell, as shown in Table 2.8.

Table 2.8 - Channel masking definition

Field	Description	LTE example
system.channels{}.id	tag identifying the channel	PSS/SSS
system.channels{}.idx	indexes if used resources in the OFDM grid	*
system.channels{}.start	1st channel transmission in number of RB	0
system.channels{}.RBrep	period of transmission in number of RB	10**

* Example shown in the scheduler description

** In LTE, the synchronization channels are repeated every 5 ms, this is 10 slot periods

For high level timing alignment, a timing structure is added containing the System Frame Number duration (SFN), defined in terms of RB duration as shown in Table 2.9. The maximum SFN (sfnMax) defines the SFN value at which the counting is wrapped.

Table 2.9 - System timing definition

Field	Description	LTE example
system.timing.sfn	duration of SFN in RB periods	20*
system.timing.sfnMax	Maximum SFN in SFN periods	1024

* In LTE 1 SFN = 1ms, this is 20 slot periods

2.2.3. *The link data structure*

The *link* data structure holds the details of transport channel processing configuration, including channel coding type, CRC size, modulation order, TTI size, as well as rate matching specific parameters. Some of the fields are updated by the framework.

The CRC definition is made through a cell containing 2 fields, the CRC size and a type. This allows for a particular CRC size to be associated with several different CRC polynomials. This is the case in LTE for the 24 bit CRC case, where two different polynomials are defined for the initial CRC attachment block and for the code block segmentation block [1]. The CRC type is a tag recognized by the CRC encoder/decoder. All LTE standardized CRC configurations with 24 bit, 16 bit and 8 bit, are supported [1].

Table 2.10 - CRC configuration

Field	Description	LTE example
link.CRCtype{ }	Cell containing CRC size and type	{24,24a}

The channel coding algorithm is a major block in the transport channel processing chain and is chosen with the *link.coding* field. By selecting a particular channel coding algorithm, the associated rate matching algorithm is implicitly also chosen.

The framework currently support 1/3 Turbo Coding and 1/3 tail-biting convolutional coding, as standardized for LTE [1].

Table 2.11 - Channel coding configuration

Field	Description	LTE example
link.coding	Channel coding algorithm	'turbo'

Modulation is defined by means of the modulation order. The supported modulations are QPSK and 16QAM, with modulation order 4 and 16, respectively.

Table 2.12 - Modulation configuration

Field	Description	LTE example
link.ModOrder	Modulation order	4

Transmission Time Interval, TTI is an important link design parameter and defines the time duration of a packet transmission. The configuration is given in multiples of RB duration, *system.RB.nsymbol*, defined in the system data structure. For the LTE PDSCH, the TTI is 1 ms, and lasts for the duration of two consecutive RBs, so that this value needs to be set to two.

Table 2.13 - TTI configuration

Field	Description	LTE example
link.tti	TTI in multiples of RB period	2*

* In LTE, for the PDSCH the TTI = 1 ms, this is 2 slot periods

The initial HARQ redundancy version to be used by the rate matching block is defined by the rv field. The meaning of this field depends on the rate matching implementation.

Table 2.14 - Initial Redundancy Version configuration

Field	Description	LTE example
link.rv	Initial redundancy version	0

The code block size to be used, may be explicitly defined, or left for empty for the framework to calculate, based on a pre-defined target code rate. If explicitly defined, it must be a valid block size, within the allowed restrictions of the channel coder.

Table 2.15 - Code Block size configuration

Field	Description	LTE example
link.CBsize	Code block size (bits)	528

The total number of available channel bits is calculated at run time by the scheduler, based on the scheduling configuration, pilot configuration and channel masks on the used TTI.

Table 2.16 - Channel size configuration

Field	Description	LTE example
link.G	Channel size (bits)	1440*

*For an allocation of 72 subcarrier, 2 TX antennas and a control region of 3 OFDM symbols

2.2.4. The UE data structure

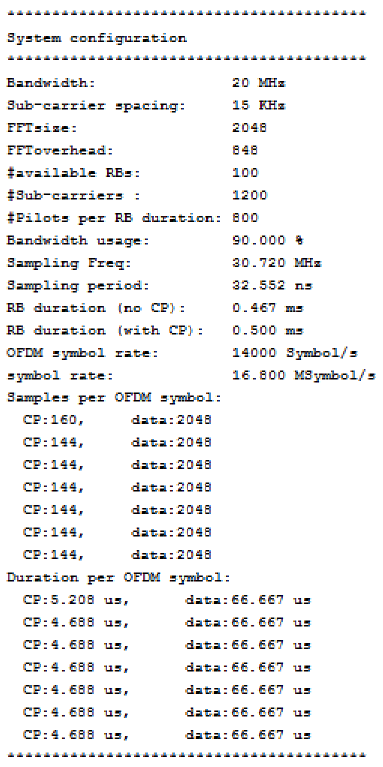
The *UE* data structure contains the UE specific configuration options, including the ones typically associated with the UE category: number of soft bits, maximum number of HARQ processes, and number or receive antennas. Currently only one receive antenna is supported. Annex A.4.3 provides the contents of the UE data structure.

2.2.5. *The run data structure*

The *run* data structure contains the collection of all configuration instances to be simulated. Annex A.4.5 provides the detailed contents of the run data structure.

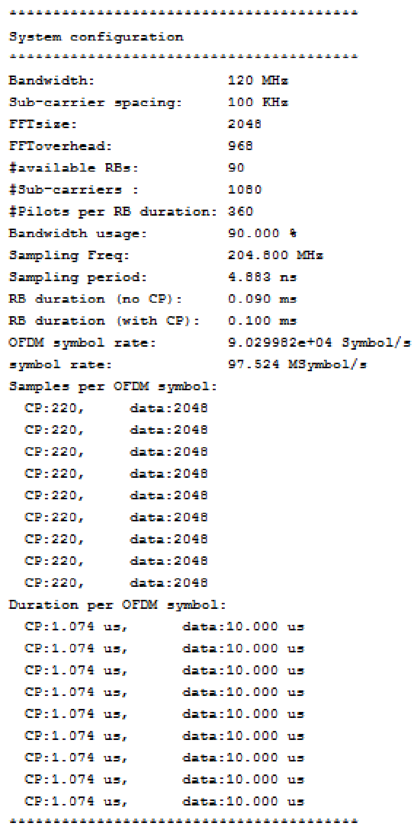
2.2.6. *Configuration examples*

Whenever the framework is initiated, a validity check will be performed on the consistency of the system configuration. If allowed by the print level, the simulation manager will print a summary of the system information to the console with potential warnings or errors. Figure 2.6 shows the system configuration summary for an LTE configuration with 20 MHz, two transmit antennas and normal cyclic prefix. Figure 2.7 shows the configuration of a hypothetical system with 120MHz bandwidth, 100 KHz subcarrier spacing, one transmit antenna, nine OFDM symbols per RB for a total RB duration of 100ms, and a very short cyclic prefix duration of approximately 1 us. This illustrates the flexibility of the framework in supporting a vast number of OFDM numerology.



```
****************************************
System configuration
****************************************
Bandwidth:                 20 MHz
Sub-carrier spacing:       15 KHz
FFTsize:                   2048
FFToverhead:               848
#available RBs:            100
#Sub-carriers :            1200
#Pilots per RB duration:   800
Bandwidth usage:           90.000 %
Sampling Freq:             30.720 MHz
Sampling period:           32.552 ns
RB duration (no CP):       0.467 ms
RB duration (with CP):     0.500 ms
OFDM symbol rate:          14000 Symbol/s
symbol rate:               16.800 MSymbol/s
Samples per OFDM symbol:
  CP:160,     data:2048
  CP:144,     data:2048
  CP:144,     data:2048
  CP:144,     data:2048
  CP:144,     data:2048
  CP:144,     data:2048
  CP:144,     data:2048
Duration per OFDM symbol:
  CP:5.208 us,      data:66.667 us
  CP:4.688 us,      data:66.667 us
  CP:4.688 us,      data:66.667 us
  CP:4.688 us,      data:66.667 us
  CP:4.688 us,      data:66.667 us
  CP:4.688 us,      data:66.667 us
  CP:4.688 us,      data:66.667 us
****************************************
```

Figure 2.6 – System information printed to the console when starting the simulation. Example of a 20 MHz LTE system configuration with normal cyclic prefix and two transmit antennas.



```
****************************************
System configuration
****************************************
Bandwidth:                120 MHz
Sub-carrier spacing:      100 KHz
FFTsize:                  2048
FFToverhead:              968
#available RBs:           90
#Sub-carriers :           1080
#Pilots per RB duration:  360
Bandwidth usage:          90.000 %
Sampling Freq:            204.800 MHz
Sampling period:          4.883 ns
RB duration (no CP):      0.090 ms
RB duration (with CP):    0.100 ms
OFDM symbol rate:         9.029982e+04 Symbol/s
symbol rate:              97.524 MSymbol/s
Samples per OFDM symbol:
  CP:220,     data:2048
  CP:220,     data:2048
  CP:220,     data:2048
  CP:220,     data:2048
  CP:220,     data:2048
  CP:220,     data:2048
  CP:220,     data:2048
  CP:220,     data:2048
  CP:220,     data:2048
Duration per OFDM symbol:
  CP:1.074 us,      data:10.000 us
  CP:1.074 us,      data:10.000 us
  CP:1.074 us,      data:10.000 us
  CP:1.074 us,      data:10.000 us
  CP:1.074 us,      data:10.000 us
  CP:1.074 us,      data:10.000 us
  CP:1.074 us,      data:10.000 us
  CP:1.074 us,      data:10.000 us
  CP:1.074 us,      data:10.000 us
****************************************
```

Figure 2.7 – System information printed to the console of a hypothetical system with 120 MHz bandwidth, 100 KHz subcarrier spacing, nine symbols per RB and a CP of ~1us per OFDM symbol.

Adding a new configuration to the framework, is done in the *systemConfig.m* module with an appropriate tag associated to the configuration. The script can then invoke this system configuration by calling the *systemConfig* API with the matching tag. LTE FDD configurations with all supported bandwidths, with normal and extended cyclic prefix, and with one and two transmit antennas are already preconfigured.

Chapter 3

The Simulation Framework implementation

The previous chapter provided an overview of the framework entities and most important data structures. This chapter focuses on the implementation aspects of the main building blocks of the simulator entity. Section 3.1- focuses on the transport channel processing related blocks. Section 3.2- presents the implementation overview for HARQ and repetition handling, while section 3.3- focuses on the scheduler.

3.1- Transport channel processing

The physical layer, having received a transport channel from the MAC layer, will process that transport channel through a series of steps, at baseband level, before sending it over the air. This is called the transport channel processing chain. Typically, the specifications will only standardize the transmitter chain, while imposing performance requirements on receiver chain.

This section describes the implementation of four different transport channel related modules. The remaining modules are described in Annex D.1 - . In this treatment, transmit and receive modules, are described in conjunction, whenever possible. Channel model related blocks are explained in D.1.5 and D.1.6, including the method for perfect channel estimation derivation.

The framework was implemented using MATLAB, a commercial simulation environment. MATLAB supports a large number of communication related algorithms, specifically within the Communications System toolbox package. These algorithms are tested and optimized for the MATLAB environment. Whenever possible, the framework makes use of system objects. In these cases, a wrapper module is created handling the configuration and communication with the relevant system object. Annex B.1 - provides a lists of all MATLAB Communications system objects used and the relevant modules.

3.1.1. The channel coding and decoding blocks

The *channelCoding.m* module receives a block of size n and outputs a block with a size m x n + t, where 1/m is the native code rate of the channel coder, and t is the number of tail bits. This module does not actually implement the channel encoding. Instead, and based on the input parameter *config.link.coding*, it will instantiate the relevant module registered to do the requested encoding operation. As illustrated on the left side of Figure 3.1, the framework includes modules for 1/3 turbo coding and 1/3 tail-biting convolutional coding as defined in [1]. Adding additional channel coding algorithms to the framework, requires the development of the coding/decoding module and registering it on *channelCoding.m* module. The second input parameter, *config.enableCofing*, allows bypassing the channel coding operation.

The turbo encoder is implemented on *ch_turboCoding.m* and is based on the MATLAB system object *comm.TurboEncoder*. The system object is configured with the polynomials and internal interleaver indices according to the LTE turbo code definitions [1]. The size of the input block is within the range of 40 to 6144 bits. However, the LTE turbo code internal interleaver, being optimized for parallelized decoding [31], only allows a subset of blocks sizes within this interval, with the granularity increasing progressively from 8 bit to 64 bit. The traffic generation module, *getNewTB.m*, will consider these size limitations when allocating a new block for turbo coding. The turbo encoder has a code rate of 1/3 and 12 tail bits, so that m is 3 and t is 12.

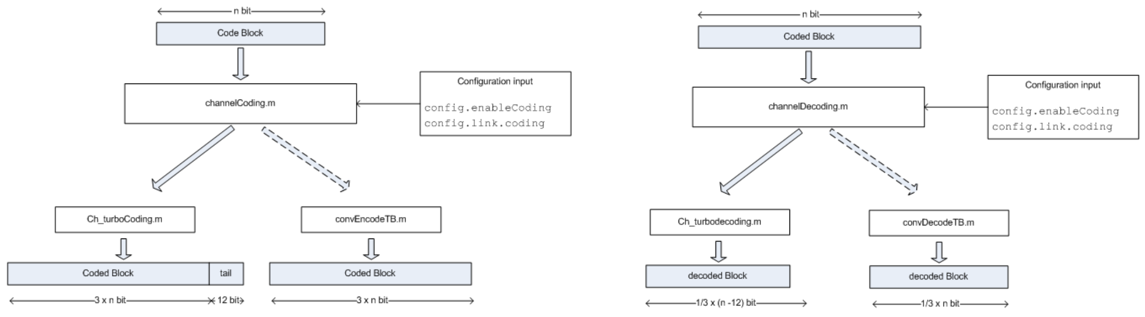


Figure 3.1 – Channel coding (right) and decoding (left).

The tail-biting convolutional code allows for an efficient tail-free encoding operation and a code rate of 1/3, thus m is 3 and t is 0. It is implemented on the *convEncodeTB.m* module, based on the MATLAB system object *comm.ConvolutionalEncoder*, with the LTE polynomials defined in [1]. To implement the tail-biting operation, the initial state of the encoder is initialized to the last 6 bits of the code block [33].

The channel decoding operation is implemented with the same concept as the encoder. The module *channelDecoding.m*, calls the relevant registered channel codding module for the decoding operation, based on the *config.link.coding* input parameter, as shown on the right side of Figure 3.1.

The turbo decoding in *ch_turbodecoding.m* relies on the MATLAB system object *comm.TurboDecoder*, with the same base configuration as the encoder. For maximum decoding performance, the decoding algorithm is set to true a posteriori probability decoding. The MATLAB implementation does not support early termination check, always using the fixed number of configured iterations. The number of decoding iterations is set to six.

The tail-biting convolutional decoding is performed in *convDecodeTB.m* based on the MATLAB system object *comm.ViterbiDecoder* with the same polynomials configuration as the encoder. Additionally, the input is configured for soft-bit input with the tail truncated. The *comm.ViterbiDecoder* system object has no native support for the tail-biting decoding operation. The decoding was implemented using the suboptimal decoding scheme approach defined in [32]. The simulated decoding performance can be found in Chapter 4.

3.1.2. The Rate Matching transmitter and receiver blocks

The rate matching stage is a fundamental block in the transport channel processing chain, matching the rate from the channel encoder to the actual channel rate available for transmission. Most common than not, these rates will be different, requiring either puncturing, when the channel rate is smaller than the coding rate, or repetition, otherwise. This is shown on the left side of Figure 3.2, where an input coded block of size n is adapted to have an output size G. The available channel size G, is calculated at run time by the scheduler and updated in the *config* data structure, on the *config.link.G* field.

Two additional major functions of the rate matcher are the operations of interleaving and redundancy version selection, when applicable.

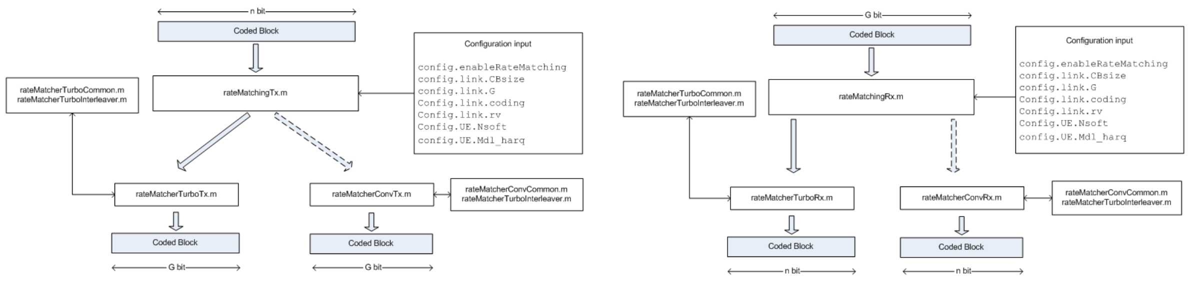


Figure 3.2 – Rate Matching transmitter (left) and receiver (right)

The framework supports the LTE rate matching blocks for turbo coding and 1/3 tail-biting convolutional code as defined in [1]. A particular rate matching implementation is likely to be optimized for a particular channel encoding scheme. Therefore, adding support for an additional channel coding algorithm, requires, in most cases, the addition of a related optimized rate matching module.

The rate matching transmitter operation for the turbo encoder is implemented in the *rateMatcherTurboTx.m* module with support of two additional modules *rateMatcherTurboCommon.m* and *rateMatcherTurboInterleaver.m*. Bypassing of rate matching is possible through the input parameter *config.enableRateMatching*.

Figure 3.3 shows an overview of the implementation. The incoming bit sequence, being turbo encoded, is composed by a sequence of triplets with one systematic bit s, followed by two parity bits, p0 and p1. The bits are separated into sub-blocks of the same type and interleaved according to a predefined permutation pattern. The output from the sub-block interleavers are then placed in a circular buffer, starting with the systematic bits, and followed by an interlaced sequence of parity bits. The output sequence is read from the circular buffer starting from one of four fixed positions defined by the redundancy version field, *config.link.rv*. For the initial transmission, the redundancy version is set to zero so that the output includes as many systematic bits as possible. Depending on the size of the output block G, the readout from the circular buffer may not contain all bits. The excluded bits are punctured. For eMTC devices, it is expected to often work with low spectral efficiency, so that G is often larger than the circular buffer size. In this case, the readout will include the same bit positions more than once.

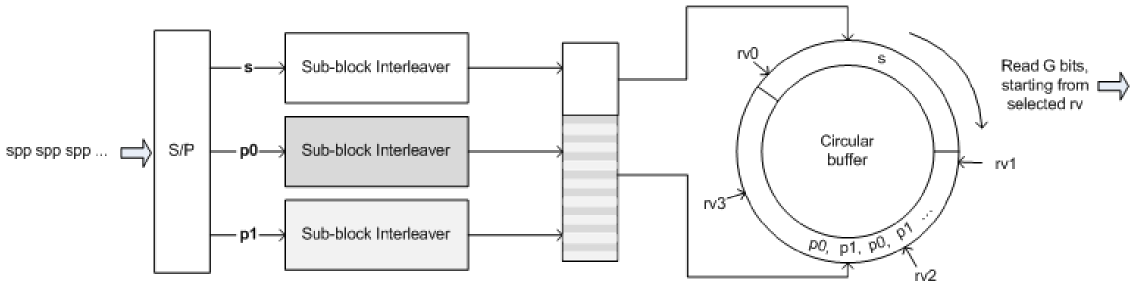


Figure 3.3 – Rate Matching detail, transmitter.

The rate matching for the convolutional coding has a similar architecture as for the turbo coding, but with some important differences. Since there are no systematic bits in the output from the convolutional coder, there is no differentiation between the three systematic output streams. Each group of bits is interleaved and place in sequence into the circular buffer. The readout is perform from the initial position, as it does not support different redundancy versions.

The receiver rate matching operation is handled by the *rateMatchingRx.m* module, selecting the correct module for the decoding algorithm, as shown on the right side of Figure 3.2.

The receiver performs the reverse operation, as described for the transmitter, rebuilding the circular buffer with the correct size, populating it with the G income bits, de-interlacing the different streams, to finally rebuild the block for channel decoding.

One difference to the transmitter, where the entire operation is performed with hard bits, at the receiver, the input block is composed of G soft-bits, representing the Log-likelihood ratio (LLR) output from the demodulator. This is an important aspect when the input block contains repetitions, since the rate matching operation can combine the repeated soft-bits while populating the circular buffer, thus increasing the received bit energy. The framework support both hard and soft-bit operation, depending on the modulator configuration.

3.1.3. The symbol mapping and de-mapping modules

The symbol mapping function populates the OFDM grid, by placing the data into the allocated OFDM resources. It also populates the pilot masks and all the allocated channels. Implementation is done in the *symbolMapping.m* module, and an overview is shown on the left side of Figure 3.4.

An empty OFDM grid is initialized for each antenna port with the time and frequency dimensions defined in the system and link data structures. For the transport channel, the mapping is based on the scheduling input parameter containing the allocation mask for each antenna port. Placement is performed on resource block basis on the increasing order of subcarrier, followed by increasing order of OFDM symbol [11].

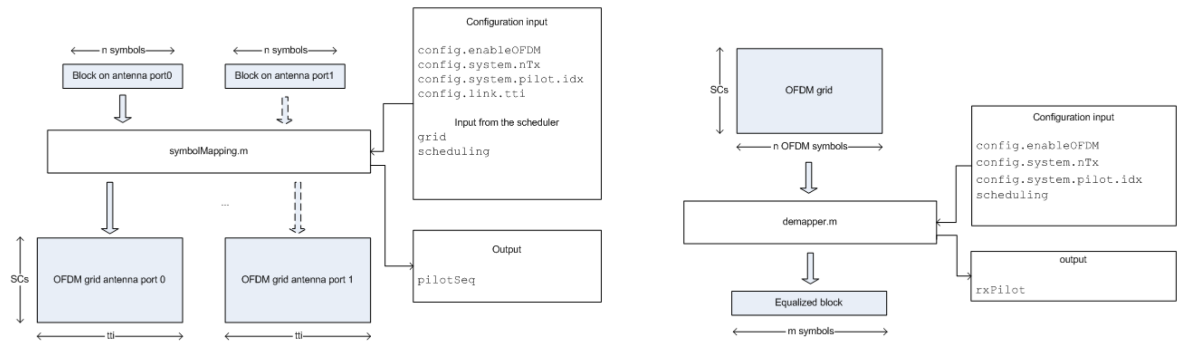


Figure 3.4 – Symbol mapper (left) and de-mapper (right)

The OFDM masks for pilot symbols are contained in the grid structure received from the scheduler, as described in 3.3.1. The symbol mapper will generate random QPSK modulated sequence for both pilots and all additional masked channels. For the masked channels, the pilot is divided across transmit antennas. The pilot sequence is returned by the module to allow pilot based channels estimation.

The *demapper.m* module removes the transport channel symbols from the OFDM grid, as per scheduling allocation, performing the reverse operation as the symbol mapper. The pilot sequence is also retrieved and the de-mapped pilot sequence is returned.

This module is disabled when OFDM is disabled in *config.enableOFDM*.

3.1.4. The OFDM encoder and decoder block

The OFDM encoder receives one OFDM grid for each transmit antenna and outputs the time domain encoded signal. The encoder is implemented in *OFDM_mod_tti.m* module as illustrated in Figure 3.5.

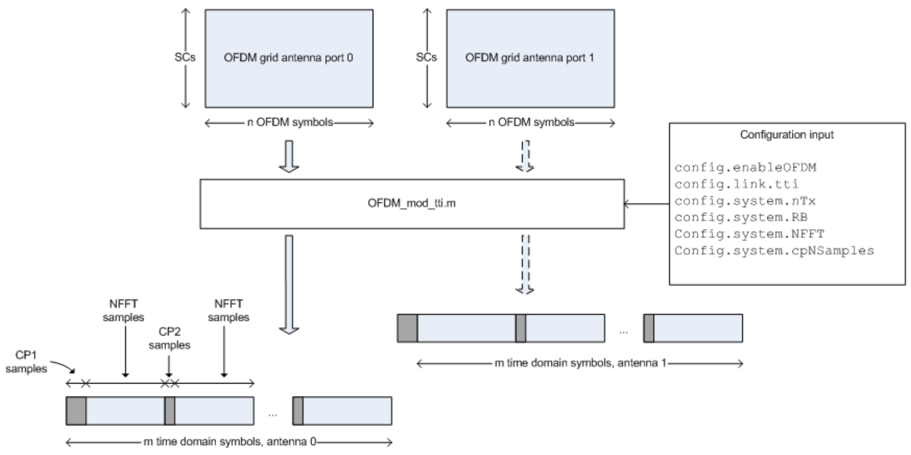


Figure 3.5 – OFDM encoder

The received grid has the number of OFDM symbols corresponding to one entire TTI, as per *config.link.tti* and *config.system.RB*. The encoding operation is performed on an OFDM symbol basis, and is performed in three steps, as illustrated on the left side of Figure 3.6. On a first step the k subcarriers within the symbol are re-arranged for the complex iFFT operation, a zero DC sub-carrier is added and finally, zero padding on the non-used external sub-carriers is added, in order to match to the chosen NFFT size, as given by *config.system.NFFT*. The second step is the iFFT operation outputting a time-domain OFDM signal with NFFT complex samples. The last step adds a cyclic-prefix, by copying the last i samples from the time-domain OFDM signal, and appending them to be beginning. The size of i is given by *config.system.cpNSamples*, and may be different across OFDM symbols.

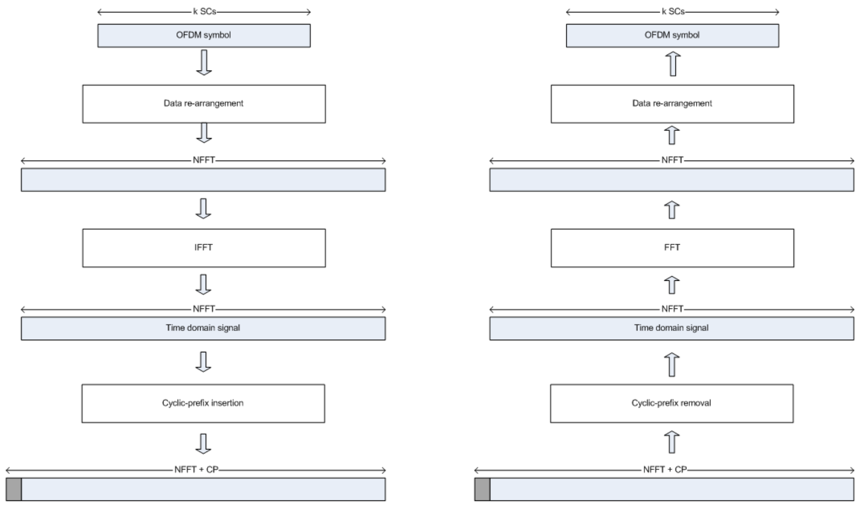


Figure 3.6 – OFDM encoder (left) and decoder (right)

The OFDM decoding operation performs the reverse operation, as illustrated on the right side of Figure 3.6. The cyclic prefix is first removed, the FFT operation converts the signal to the frequency domain, and finally, the signal is filtered to the k center subcarriers and the DC carrier is removed.

3.2- HARQ and repetitions

Repetition and retransmission of packets is extensively used in the physical layer of LTE, therefore requiring efficient combining methods. This is especially true for 3GPP release 13 eMTC devices operating in coverage enhancement mode, where a high repetition count, of up to 2048, is possible [2].

Analysis of the different implementations as well as simulation results are presented in sections 4.2 - and 4.3 - . This section presents the framework implementation of the supported combining options. Figure 3.7 illustrates part of the receiver chain, showing the different stages where combining may be implemented. There are two main combining options, bit-level and symbol-level combining. While symbol-level combining can be done earlier within the receiver chain, requiring less processing power, bit-level combining is required when the packets have different redundancy versions (RV).

Bit level combining happens in two stages, during and after the receive rate matching operation. In those cases where the code rate of the transmitted packet is lower than the native channel encoder code rate, each packet will already contain repeated bits. These are combined by the rate matching receiver, as described in section 3.1.2. Combining of repeated blocks is performed after rate matching by adding the corresponding soft bits.

Symbol level combining, can be done at different levels. The post-equalization method allows for Maximum Ratio Combining (MRC), where the channel gains can be used as weighting factors, therefore maximizing SNR [43]. The framework implements this option by setting *config.CombinePreEq* to 0. The channel weight for each repetition is calculated based on the average channel gain on allocated resources.

The second symbol level combining method, post-FFT and pre-equalization method, allows for a simpler implementation, with the combining weight being equal for all repetitions. The framework performs this combining method by setting *config.CombinePreEq* to 1.

The third option listed in Figure 3.7, of combining prior to OFDM demodulation, is not currently implemented. This option, by combing the time domain signals, requires only one FFT operation for the combined set, as opposed to the other combining methods, requiring one FFT operation per repetition [43].

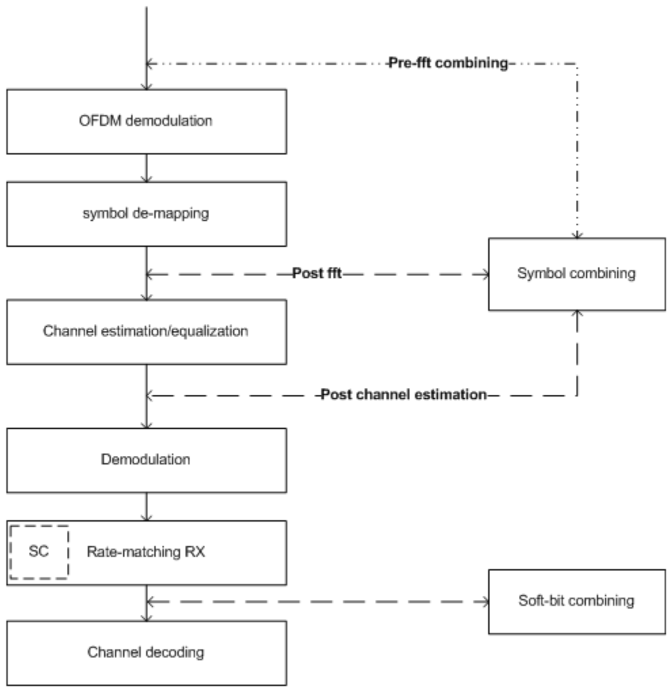


Figure 3.7 – Combining methods at the receiver

Both HARQ and repetition functionality may be enabled or disabled as per configuration of the Boolean settings *config.enableHARQ* and *config.enableRepetition*, respectively. The maximum number of repetitions and retransmissions are defined by the settings *config.maxRepetitions* and *config.maxHarqRetx*.

When repetition is enabled, it is possible to configure an RV period so that consecutive repetitions within this period have the same RV, as illustrated in Figure 3.8. This allows for symbol level combining, during the RV period, followed by bit level combining, at the end of each RV period. The RV period is configurable through the parameter *config.RVperiod*.

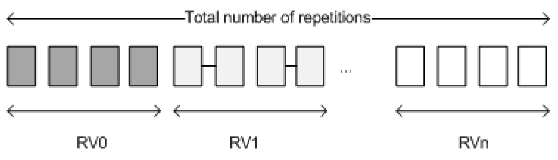


Figure 3.8 – RV period concept

While keeping the RV pattern, it is also possible to enforce the same RV on all RV periods, without changing the combining methods, by setting the input *config* parameter *config.forceHarqChaseComb* to true.

3.3- The scheduler, channel masking and timing

The scheduler is an important module in the framework. It is responsible for managing all OFDM resources on a TTI basis. This includes mapping of all masked channels to a particular location in the OFDM grid, and assigning resourced to the transport channel. The scheduler does not populate the grid with any data, it will rather provide the allocations mask for each channel to be included on the next scheduling period. The scheduler is implemented in the *scheduler.m* module.

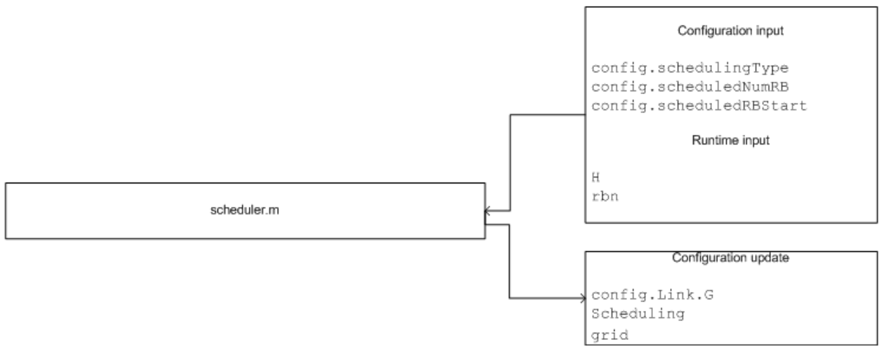


Figure 3.9 – The scheduler module

3.3.1. OFDM grid Initialization and channel masking

As discussed in section 2.2.2, the system data structure allows for a fully flexible OFDM configuration. This includes, not only the OFDM numerology, but also the minimum resource block allocation configuration, pilot structure configuration as well as channel masking for any number of channels. It is also possible to associate each channel to a particular time pattern. The configurability of the Transmission Time Interval (TTI) in *config.link.tti*, described in 2.2.3, adds further flexibility to the framework.

All this information is used by the scheduler when initializing the OFDM resources. The scheduler is started in the beginning of every new TTI, and immediately after the timing module has updated the *RB* counter, *RBn*. The initialization principles are based on the flow chart shown in Figure 3.10. The scheduler starts by building one OFDM grid for each antenna port, for the entire system bandwidth, and for the duration of one TTI. Next, the pilot positions are defined. The pilot symbol indexes from *config.system.pilot.idx*, defined for one Resource Block (RB) period, need to be repeated for the entire TTI and for each of the antenna ports. The pilot positions on any antenna port will mask the relevant Resource Element (RE) for all antenna ports. This is crucial to allow channel estimation across multiple antennas.

The next step is to iterate through the configured channels in *config.system.channels*, and evaluate if, based on the channel timing configuration, *config.system.channels{}.RBrep* and *system.channels{}.start*, it should be included for scheduling on the next TTI. The channels selected for allocation, will be added sequentially based on the defined OFDM mapping *config.system.channels{}.idx*. To simplify the channel masking configuration, the allocation is defined, not accounting for any pilot symbols. It is the task of the scheduler to mask out the pilot positions, when overlapping with any channel.

Having mapped all pilots and valid channels, the remaining resources are available for the channel being handled by the transport processing chain. These are resources that the scheduler can now allocate based on the chosen bandwidth and scheduling strategy. The entire resource allocation is kept in the *grid* data structure.

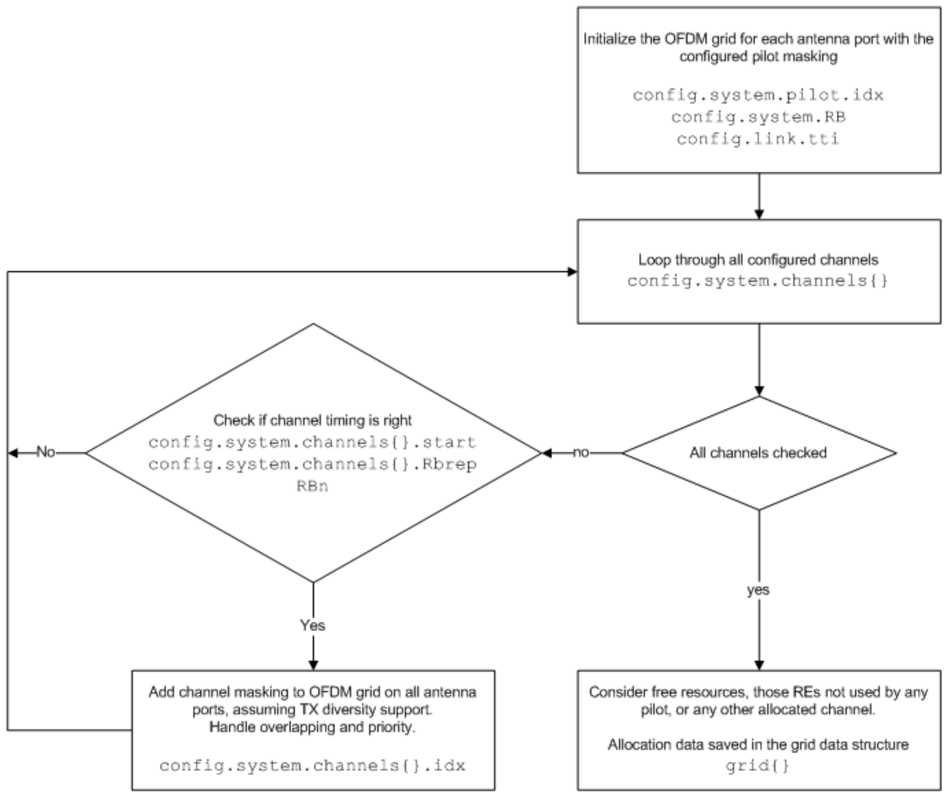

Initialize the OFDM grid for each antenna port with the configured pilot masking
config.system.pilot.idx
config.system.RB
config.link.tti
Loop through all configured channels
config.system.channels{}
All channels checked
no
Check if channel timing is right
config.system.channels{}.start
config.system.channels{}.Rbrep
RBn
No
Yes
yes
Add channel masking to OFDM grid on all antenna ports, assuming TX diversity support. Handle overlapping and priority.
config.system.channels{}.idx
Consider free resources, those REs not used by any pilot, or any other allocated channel.
Allocation data saved in the grid data structure
grid{}

Figure 3.10 – OFDM grid initialization for next TTI.

The channel masking is a flexible configuration tool and may be used for other purposes than signaling physical channel masking. It may also be used to create particular traffic patterns masks in order to mimic a certain load scenario in the user plane.

3.3.2. OFDM grid Initialization example for LTE

An example of an OFDM grid mapping is shown in Figure 3.11 based on an LTE configuration with normal cyclic prefix (14 OFDM symbols per TTI), 3 MHz bandwidth and two transmit antennas. The framework includes a module, *plotOFDMgrid.m*, for plotting the allocated resources to each channel, on a TTI basis, as shown in Figure 3.11.

The pilot symbols, in LTE called Cell-specific Reference Signals (CRS), are configured, on OFDM symbols zero, four, seven and eleven, with a hexagon pattern. The pattern is shifted by three subcarriers between the two antennas [11].

Three additional channel masks are configured in the *config.system.channels* in order to mimic the typical signaling overhead of an LTE cell. The first mask contains the three physical channels that are part of the, so called, control region. The control region is typically spanning the first two or three OFDM symbols on every TTI, and the entire system bandwidth. In the example from Figure 3.11, the size of control region is set to three OFDM symbols. The channels included in this mask are the following:

- The Physical Control Format Indicator Channel (PCFICH), a very low bit rate channel indicating the size, in OFDM symbols, of the control region [11].
- The Physical Hybrid-ARQ Indicator Channel (PHICH), used to carry the downlink HARQ acknowledgement messages for received uplink transmissions [11].
- The Physical Downlink Control Channel (PDCCH), used to carry the scheduling commands to the terminals [11].

For 3GPP release 13 compliant category M1 eMTC devices, the receiver bandwidth is limited to 1.4 MHz, thus, the control region cannot be decoded by these devices. Nevertheless, in order to serve all other devices, the control region persists and needs to be taken into account when simulator M1 devices.

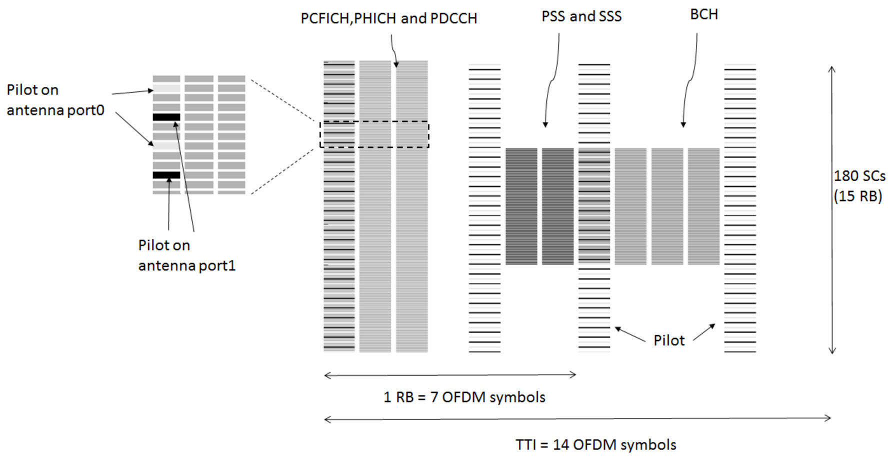


Figure 3.11 – Channel masking example

The second mask contains the Primary and Secondary Synchronization Signals, PSS and SSS [11]. These are used for frame synchronization, support in detecting the physical cell identity of the cell, and the duplex operation mode, TDD or FDD. PSS is sent on the 6th OFDM symbol, and SSS on the 7th, and they span the center 72 subcarriers. PSS and SSS are transmitted with a period of five TTIs.

The third and last mask contains the Broadcast Channel (BCH) [11], carrying part of higher layer System Information. Only a limited subset of system information, the Master System Information (MIB), is sent though the BCH, with the largest part being sent over the user plane [12]. The MIB contains information on system timing, frequency bandwidth and number of transmit antennas configured. The BCH is sent with a period of 20 *RBn*, on the 1st four OFDM symbols, starting from the 2nd *RBn*. Just as the PSS and SSS, it spans the center 72 subcarriers.

Figure 3.12 illustrates the OFDM grid allocations on antenna port 1, following the scheduler initialization process for the configured LTE system on three different TTIs.

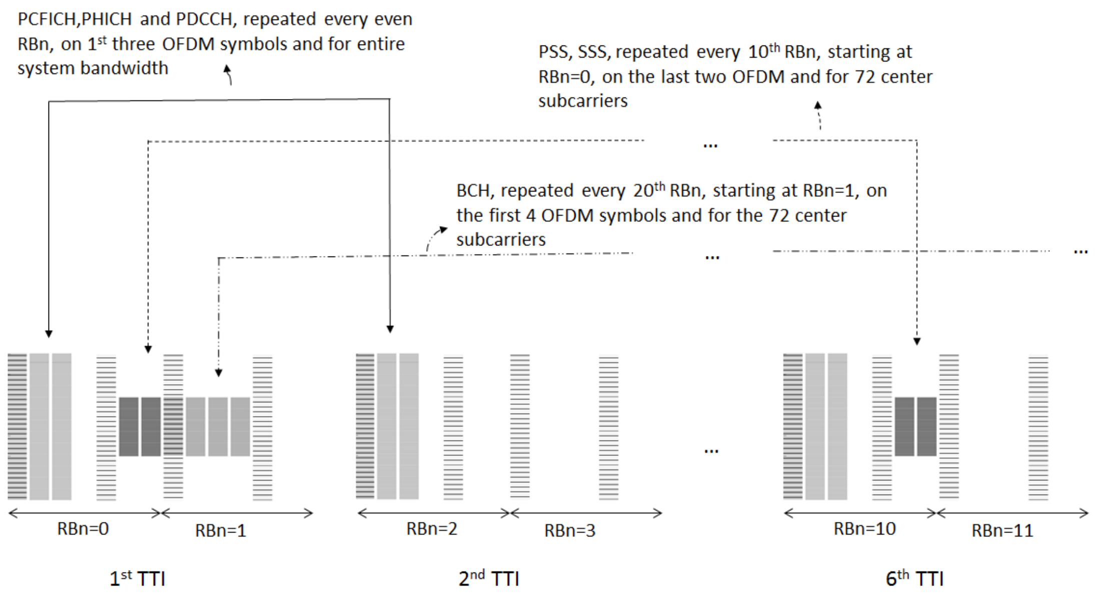

PCFICH,PHICH and PDCCH, repeated every even RBn, on 1st three OFDM symbols and for entire system bandwidth
PSS, SSS, repeated every 10th RBn, starting at RBn=0, on the last two OFDM and for 72 center subcarriers
BCH, repeated every 20th RBn, starting at RBn=1, on the first 4 OFDM symbols and for the 72 center subcarriers
RBn=0
RBn=1
RBn=2
RBn=3
RBn=10
RBn=11
1st TTI
2nd TTI
6th TTI

Figure 3.12 - Channel masking timing.

The white area in Figure 3.11, represents the available resources for user data scheduling. Assigning user data resources is the next task of the scheduler.

3.3.3. Transport channel resource allocation

After the OFDM initialization process, the scheduler is aware of the total available resources for transport channel allocation. The next step is to allocate a subset of these resources to the target transport channel.

The total available bandwidth is divided into contiguous frequency slots, each with the size of one RB, *system.RB.nSC*. The selection of the exact frequency slots to allocate is based on the chosen scheduling algorithms defined by the property *config.schedulingType*. The framework currently supports three algorithms:

- ‘Fixed-localized’, where the scheduler allocates a fixed amount of contiguous resources on a precise frequency location, given by the properties *scheduledRBStart* and *config.scheduledNumRB*. These properties specify the beginning and size of allocation in number of RBs.
- 'Fixed-hopping', where a fixed amount of contiguous resources, defined by *config.scheduledNumRB* are allocated randomly by the scheduler at each scheduling interval. The location of the allocated resources will hop across randomly selected frequency slots in an attempt to provide frequency diversity between successive scheduling intervals. Especially for slow varying channels.
- 'Fixed-hopping-hmax', where the scheduler attempts to allocate a fixed amount of contiguous resources, defined by *config.scheduledNumRB*, on the

slots with highest downlink channel gains. To do this, the scheduler is assumed to have knowledge of the downlink channel gains from previous transmissions. The channel gains are averaged across antennas and across frequency slots, and each possible allocation interval of size *config.scheduledNumRB*, is given a corresponding channel weight. The scheduler then assigns the allocation with highest weight. Depending on the downlink channel rate change, consecutive scheduling intervals, may therefore have the same of different allocations.

The effective allocated resources are returned by the *scheduler.m* module in the scheduling parameter. Additionally, the scheduler will also update the allocated channel size in *config.link.G*. This will be used by the rate matching block to calculate the amount of puncturing or repetition needed.

Simulation results for the different algorithms are presented in 4.4 - .

Chapter 4

Simulation results

This chapter presents some simulation results in an attempt to illustrate the framework flexibility, as well as to validate key components within the framework, both against theoretical results, as well as results published by other sources.

4.1 - Module validation

4.1.1. Un-coded and coded performance evaluation

The framework offers the possibility to bypass particular modules, thus, allowing evaluation of specific components, as well as simplifying the introduction of additional modules. Table 4.1 lists a configuration used to evaluate performance of un-coded modulation signals over OFDM. The configuration bypasses the modulator and associated rate-matching modules, forces a fixed code rate of 1, and finally instruct the modulator to provide hard-bit outputs. The results for un-coded 16QAM and QPSK are shown in the left chart of Figure 4.1 compared to the theoretical results [42].

Table 4.1 — Simulator configuration for module validation example

Parameter	Value
config.enableCoding	false
config.enableModulation	true
config.enableSoftDemodOut	false
config.enableRateMatching	false
config.enableFixedCodeRate	1

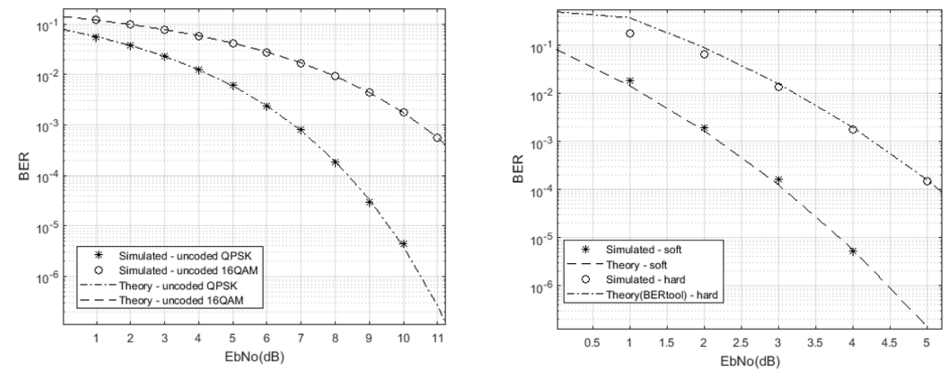


Figure 4.1 Un-coded QPSK/16QAM (left) and coded 1/3 convolutions code (right). The simulation results match the expected theoretical results.

Similarly, evaluation of channel decoding is possible for both hard and soft-bit inputs, if supported by the decoder. This can be done on any configuration by toggling the configuration setting *config.enableSoftDemodOut*. This is illustrated in the right chart in Figure 4.1 where 1/3 Tail-biting convolutional decoder performance is compared to the theoretical results. This can be done on any configuration

4.2 - HARQ

This section investigates the principles of Hybrid Automatic Repeat Request (HARQ), analysing the performance and impact of different combining strategies through simulation results.

4.2.1. Overview

HARQ is an effective way to improve link performance, by enabling the transmitter to quickly resend badly received packet, allowing the receiver to combine several packets before channel decoding, thus improving the decoding performance [38]. This requires that both the transmitter and receiver need to be able to buffer the packet, the transmitter, typically at MAC layer, the receiver at the physical layer, before channel decoding. For every sent packet, HARQ relies on a fast acknowledgment from the receiver in order to decide if a retransmission is needed. To avoid any lag while waiting for the acknowledgment, several parallel HARQ instances, called HARQ processes, may be configured, so that on successive transmission times, different HARQ processes may be used. The framework currently supports one single HARQ process, thus modeling an instant feedback, at the transmitter.

Considering the simulation configuration in Table 4.2, the simulation results are shown in Figure 4.2, illustrating the typical BLER curve for a simulated HARQ process [36].

Table 4.2 — Simulator configuration for HARQ validation

Parameter	**Value**
System bandwidth (MHz)	5
Allocation size (#PRBs)	6
Modulation	QPSK
Channel Model	AWGN
HARQ	IR
Effective code rate	1/3
Channel coding	turbo

For large SNR values, where one unique transmission can be correctly decode, HARQ has obviously no effect. As SNR decreases, a single transmission would lead to a BLER close to 100%. However, combining the initial transmission with a second retransmission provides the channel decoder with a better input, enough to successfully decode the packet. With two retransmission needed to successfully decode a packet, the effective BLER is 50%.

As SNR decreases even further, more re-transmissions are needed in order to successfully decode received packets. The overall BLER, and consequently the effective code rate, spectral efficiency, and ultimately the data rate, will decrease with increasing number of re-transmissions.

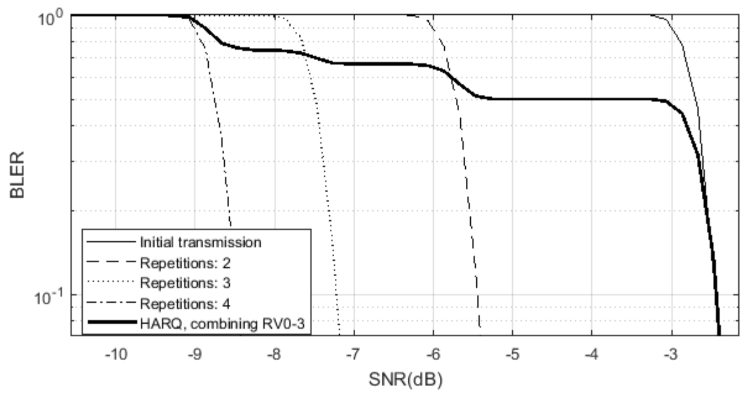


Figure 4.2 Simulation of different repetitions and HARQ effect.

4.2.2. Incremental redundancy vs chase combining

In a HARQ implementation, there are two possible main strategies, based on the contents of the retransmitted packet, Chase Combining (CC) and Incremental redundancy (IR). In Chase Combining, all retransmissions are exactly the same, allowing for simpler combining implementations, typically at symbol level, and prior to demodulation. On the other hand, with Incremental redundancy, each transmission contains a different version of original packet, typically different puncturing patterns, for higher code rates. Implementation is more complex, typically requiring bit level combining in order to reconstruct the puncturing pattern. The rate matching stage implemented in the framework for the turbo code supports both IR and CC. The convolution implementation supports only supports CC.

IR has an advantage over CC, but only for higher code rates [37], where a higher level of puncturing is present. This is confirmed by the simulation results in Figure 4.3 and Figure 4.4. The simulation was run with the same configuration from Table 4.2, for both CC and IR, but with a code rate of 0.6 and 0.2, respectively.

While at the lower code rate of 0.2, IR and CC are indistinguishable, at the higher code rate there is a gain of approximately 0.6 dB with IR on the 1st retransmission, and close to 1 dB on the 2nd retransmission.

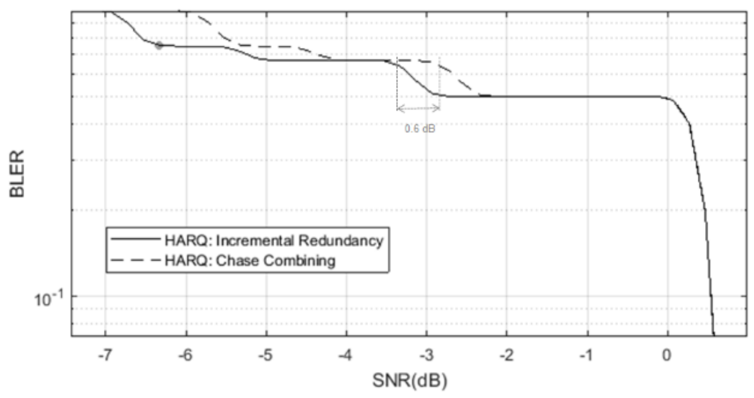


Figure 4.3 Chase combining vs Incremental redundancy, QPSK, 0.6 code rate.

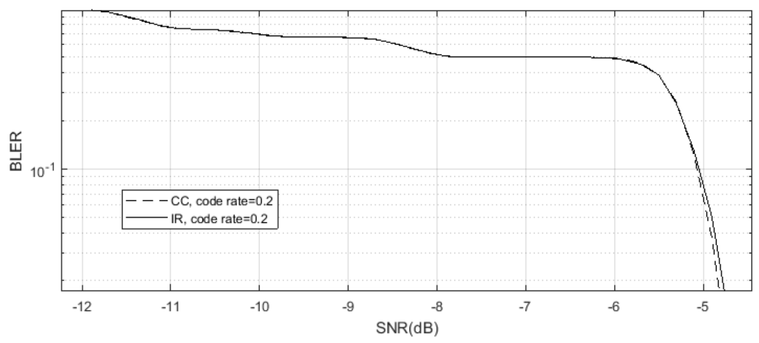


Figure 4.4 Chase combining vs Incremental redundancy, QPSK, 0.2 code rate.

Figure 4.5 shows an additional simulation results using the same configuration from Figure 4.4, but with 16QAM instead of QPSK. The results show that for higher order modulations, even at low code rates, IR has advantages over CC [37].

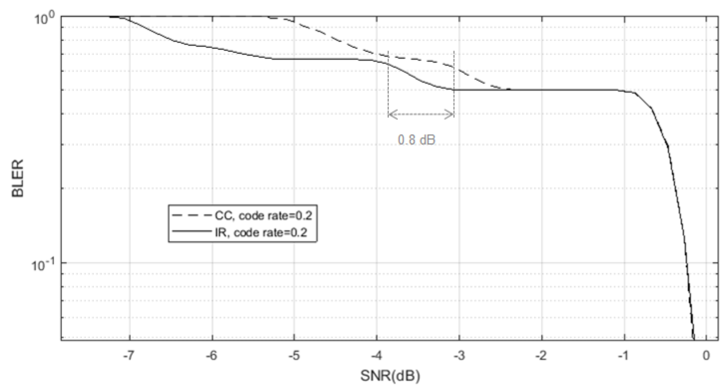


Figure 4.5 Chase combining vs Incremental redundancy, 16QAM, 0.2 code rate.

4.2.3. Bit level vs Symbol level chase combining

When combining packets with the same redundancy version (CC), the receiver has the option to perform the combining either at bit level or symbol level. Both options are supported by the framework as illustrated next.

The same base simulator configuration from Table 4.2, with the changes listed in Table 4.3, is used to simulate the combining of eight packets with the same RV. Using the parameter *config.RVperiod* set to the extreme values of 1 and 8 enforces bit level combining and symbol level combining, respectively.

Table 4.3 — Simulator configuration for bit and symbol level combining

Parameter	**Value**
config.maxRepetitions	8
config.RVperiod	1 and 8
config.forceHarqChaseComb	true
Modulation	QPSK and 16QAM
Code rate	0.6

Figure 4.6 shows that, for QPSK, there is no difference between the combining methods. However, when 16QAM is used, there is a considerable degrade, especially for higher number of repetitions, as shown in Figure 4.7. This is in line with the results in [40].

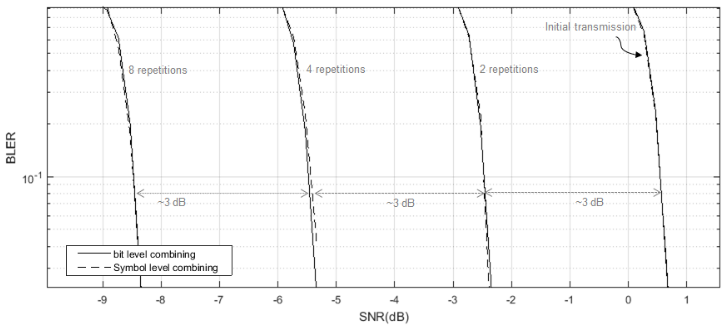


Figure 4.6 Bit level vs symbol level Chase Combining, QPSK, 0.6 code rate.

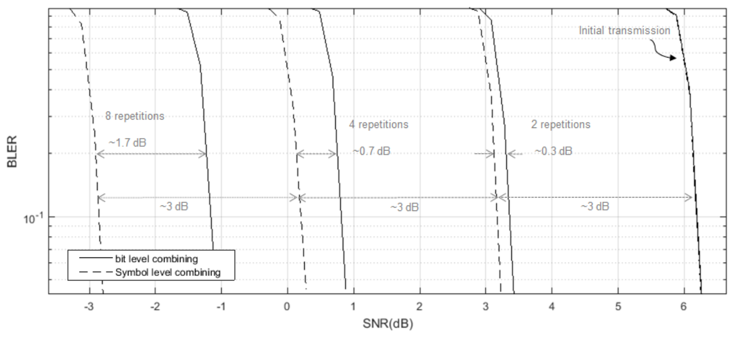


Figure 4.7 Bit level vs symbol level Chase Combining, 16QAM, 0.6 code rate.

4.3 - Repetitions

This section continues the analysis initiated in the previous section on HARQ, extending it to the case where a large number of repetition bundling is needed. Unlike HARQ, where a retransmission is based on the receiver feedback, with repetitions, a predefined number of transmissions is sent for the same packet. The receiver combines all the different repetitions, thus improving the effective SNR.

4.3.1. Overview

In order to meet the target coverage enhancements required for eMTC devices, release 13 of 3GPP specifications added support for repetition. The impact and benefit was studied in [3]. The obvious consequence is that coverage enhancement is done at the expense of spectral efficiency.

In ideal conditions the gain in SNR is linear, improving SNR by 3 dB with every doubling in the number of repetitions, but in practice, channel estimation reliability will limit these gains for increasing number of repetitions [41].

Considering a 5 MHz LTE cell over a non-faded AWGN as per configuration given in Table 4.4, the simulated SNR curves are shown tor the initial transmission and for a number of repetitions, 2,4,8,16,32,64 and 128. All repetition are sent with the same redundancy version. The gain in SNR, as expected, is approximately 3 dB per doubling in number of repetitions, as shown in Figure 4.8.

Table 4.4 — Simulator configuration for repetition

Parameter	**Value**
System bandwidth (MHz)	5
Allocation size (#PRBs)	6
Modulation	QPSK
Channel Model	AWGN
Effective code rate	1/5
Channel coding	1/3 turbo

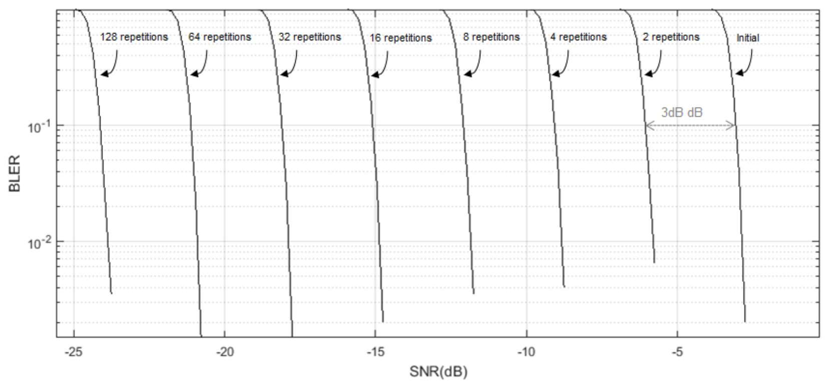


Figure 4.8 Repetitions with QPSK and symbol combining, code rate 0.2, and perfect synchronization and channel estimation

In these ideal conditions, for a number of repetitions, r, the overall SNR gain, G in dB, can be given by:

$$G = 3\mathrm{n}\,, \tag{5.1}$$

where n represents the number of times repetitions, r, were doubled, and can be given by:

$$n = \log_2 r\,. \tag{5.2}$$

The required number of repetitions to obtain a target gain G, can then be given as:

$$r = 2^{\left(\frac{G}{3}\right)}\,, \tag{5.3}$$

showing the exponential relation between gain and required repetitions. This relation is illustrated in Figure 4.9, together with the points from Figure 4.8, having 10% BLER. For high target gains, a significant number of repetitions is needed for only a modest improvement in SNR.

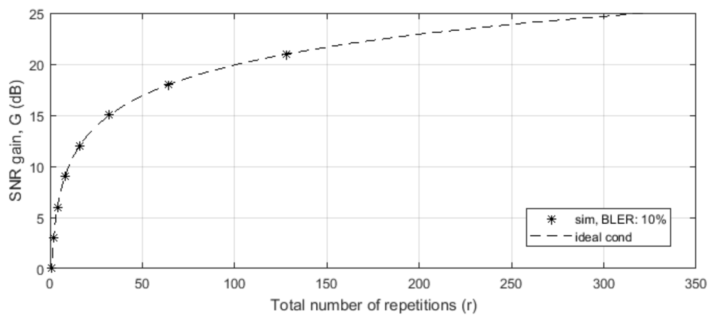


Figure 4.9 SNR gain with increasing number of repetitions, at 10% BLER

In practice, channel estimation reliability and Carrier Frequency Offset (CFO) will limit these gains for increasing number of repetitions [41] as shown in [65][9]. Having only perfect channel estimation and synchronization, the framework can currently not validate the repetition at high target gains.

4.3.2. Repetitions with different redundancy versions

Similarly to the HARQ concept, different repetitions of the same packet, may be sent with different Redundancy Versions (RV). It is possible to define a redundancy version period, so that, consecutive repetitions within the same period, all have the same RV. This is especially advantageous for high code rates with high level of puncturing.

A simulation investigating this effect is shown in Figure 4.11. The simulation is based on Table 4.4, with the changes described in Table 4.5.

Table 4.5 — Simulator configuration for RV pattern cycling

Parameter	Value
Repetitions	8
RV period	2 and 8
RV pattern	0,1,2,3
Code rate	0.6 and 0.2

The simulation compares two different RV period patterns, with a fixed number of 8 repetitions as illustrated in Figure 4.10. On pattern a), the RV period is two, so that every two consecutive repetitions have the same RV. On pattern b), the RV period is the same as the total number of repetitions, therefore all repetitions have the same RV. These patterns are tested with code rates of 1/5, with no puncturing, and 3/5, requiring a high level of puncturing. The simulation results are presented in Figure 4.11, showing approximately 1 dB gain when pattern a) on the high code rate scenario. There is only a marginal gain with the low code rate of 1/5.

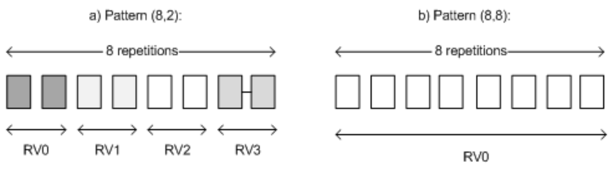


Figure 4.10 Repetitions patterns (8,2) and (8,8)

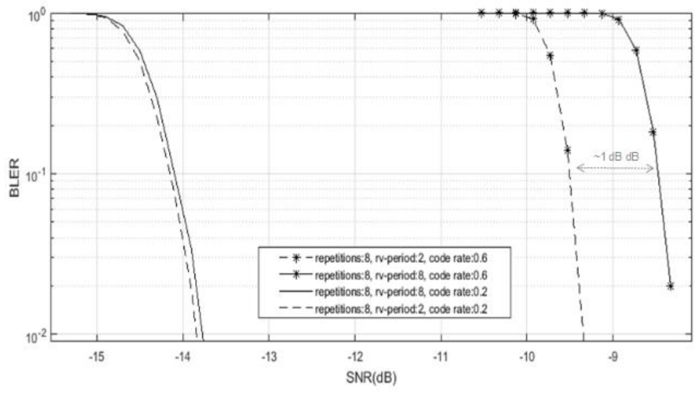


Figure 4.11 Repetitions with different RV vs. no repetitions with RV, at code rate 0.6

As mentioned in section 4.2 - , combining packets with different RVs, requires reconstruction of the puncturing or repetition pattern, thus, requiring bit level combining, after demodulation. However, for coverage enhancement where low coding rates are used, there is no expected performance improvement [15].

4.4 - Frequency hopping

This section presents a comparison of the different hopping algorithms, as described in 3.3.3. Table 4.6 describes the simulator configuration.

The configuration with no hoping, corresponding to the *'fixed-localized'* scheduling option, is configured at the lower edge of the available bandwidth. The simulation results for the 5MHz and 20 MHz simulation are shown in Figure 4.12 and Figure 4.13, respectively. As expected, the benefit of frequency hopping increases with the system bandwidth, allowing a higher degree of frequency diversity. The '*fixed-hopping*', algorithm, blindly allocating a random frequency slot, performs only modestly, when compared to the *'fixed-hopping-hmax'* algorithm which allocates resources on the frequency slot with highest channel gain. In practice, the downlink channel conditions on non-allocated resource is difficult to predict, especially in FDD systems, but it shows that any downlink channel knowledge, may effectively be used by the scheduler.

Table 4.6 — Simulator configuration for frequency hopping

Parameter	**Value**
System bandwidth (MHz)	5/20
Allocation size (#PRBs)	6
Modulation	QPSK
Antenna Configuration	2x1
Antenna correlation	low
Channel Model	EPA-5

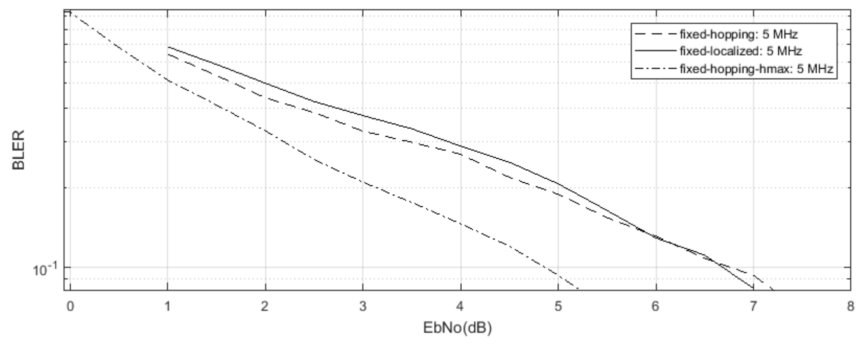


Figure 4.12 Simulation of different scheduling algorithms with 5MHz bandwidth on EPA-5 channel.

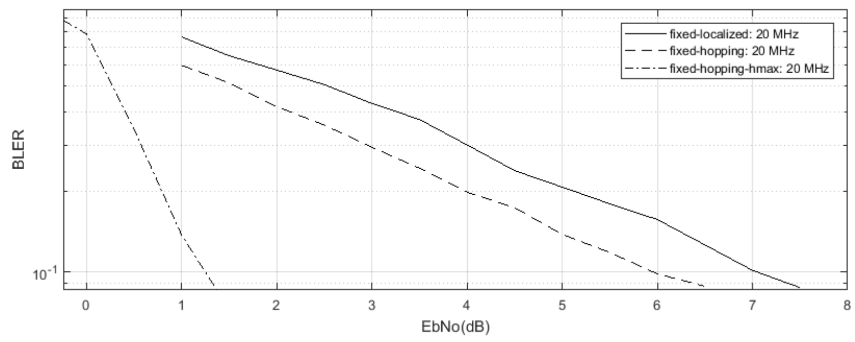


Figure 4.13 Simulation of different scheduling algorithms with 20MHz bandwidth on EPA-5.

4.5 - Coverage Enhancement improvements for eMTC

This section analysis strategies to enhance downlink coverage for eMTC devices based on the framework simulation results.

4.5.1. The importance of small code block sizes

Considering a 10 MHz LTE FDD configuration with a static allocation of 6 PRBs, QPSK modulation, 2 TX antennas and a control region of 3 OFDM symbols, the number of available channel bits per 1ms TTI is 1440 bits. The base simulator configuration is given in Table 4.7.

Table 4.7 — Simulator configuration for block size comparision

Parameter	**Value**
System bandwidth (MHz)	10
Allocation size (#PRBs)	6
Modulation	QPSK
Antenna Configuration	2x1
Antenna correlation	low
Channel Model	EPA
Doppler spread (Hz)	1
HARQ	Off
Frequency hopping	Off
Size of PDCCH region	3

With the fixed static channel allocation, the size of the code block will define the effective code rate. Figure 4.14 illustrates the simulated BLER versus SNR curves for a number of different code block sizes from 40 to 960 bits, representing effective coding rate from approximately 0.028 to 0.667.

For any given BLER, smaller block sizes require lower SNR than larger ones. It is also obvious that the gap between SNR curves increases as code block sizes get smaller hinting to the non-linear relation between SNR and code block size, for small block sizes. On current 3GPP releases, the very low block sizes, even though allowed by the Turbo coder block, are not allowed to be scheduled for larger allocation sizes [2]. Assigning small blocks has an obvious negative impact on spectral efficiency, however, for eMTC devices, spectral efficiency needs to be sacrificed in order to meet the target receiver cost and coverage improvements [3].

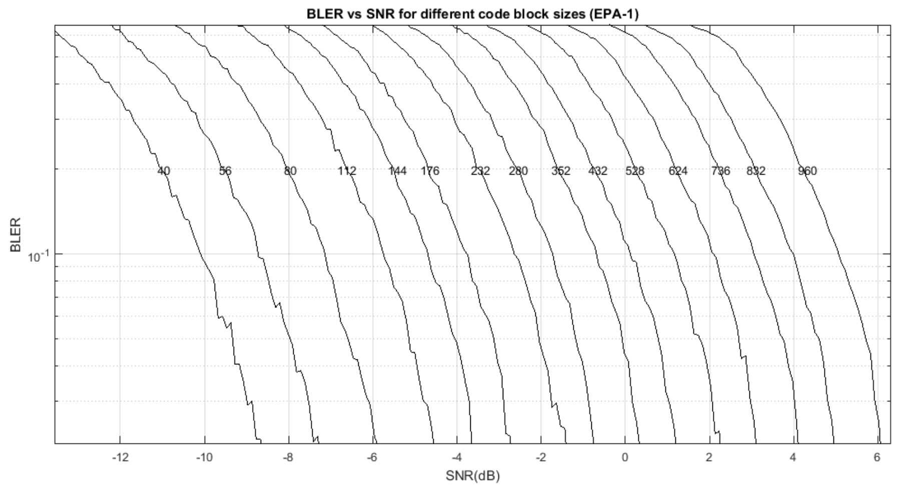


Figure 4.14 – Required SNR for different code block sizes, given a fixed allocation of 1440 channel bits.

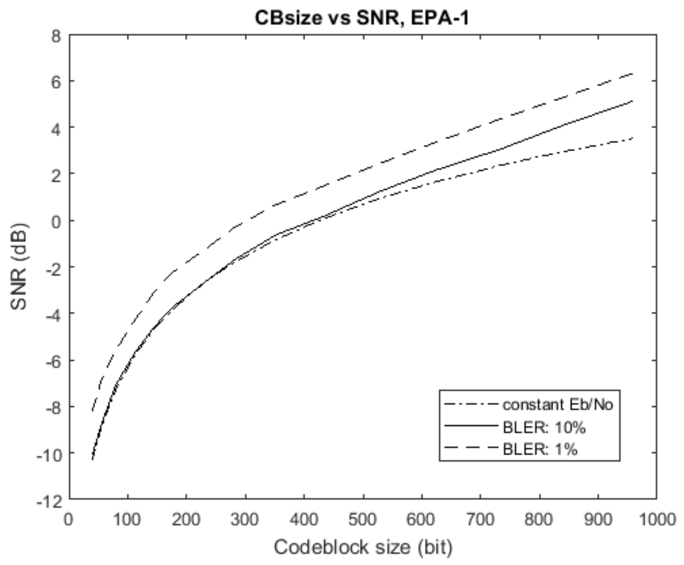


Figure 4.15 - Required SNR for different code block sizes, given a fixed allocation of 1440 channel bits and a target BLER of 10% and 1%.

Further treatment of the data from Figure 4.14 is shown in Figure 4.15, considering only the points with BLER of 1% and 10%. The resulting curves of BLER versus code block size show two distinct regions: for code blocks above 350 bits, the SNR changes linearly with an increase of the block size and at a rate of approximately 1 dBs per 100 bits. On the other hand, for block sizes below 350 bits, any changes in the code block size has a more dramatic impact in the SNR.

Generically, the relation between SNR and Eb/N0 for a given data rate can be given by [26]:

$$SNR = \frac{E_b}{N_0}\gamma \, , \tag{5.4}$$

where γ is the spectral efficiency measured in bits/second/Hz. Assuming that the coding and rate matching stages would lead to the same Eb/N0 performance, and noting that in our simulation scenario the spectral efficiency is only depending on the code block sizes, then, the gain in SNR in dB, ΔSNR_{nm}, between any two block sizes Cn, Cm can be given by:

$$\Delta SNR_{nm} = 10\log\left[\left(\frac{SNR_n}{SNR_m}\right)\right] = 10\log\left(\frac{C_n}{C_m}\right) \, , \tag{5.5}$$

Figure 4.16 includes the ΔSNR curve, referenced to Cn = 232 bits. It shows that the simulated SNR largely matches the ΔSNR, especially for smaller block sizes. In practice, however, Eb/N0 for a particular BLER is not constant across all block sizes. This is shown in Figure 4.16 where, for 10% BLER, a fairly constant Eb/N0 is reached for code rates below 1/3, but, due to puncturing on larger blocks, there is a degradation in Eb/N0 performance as code rates increases. For a target BLER of 1%, also the smaller blocks start to experience a degradation in Eb/N0 performance due to the low turbo coding performance at these very small code blocks sizes.

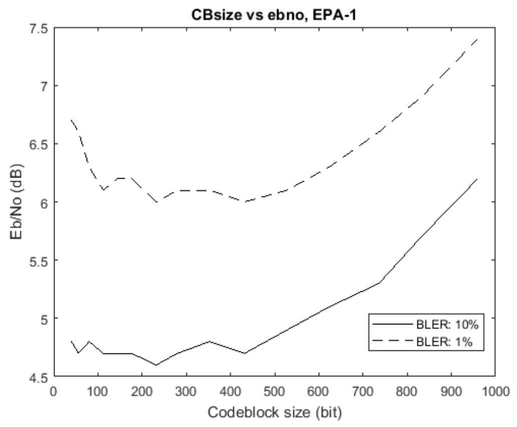


Figure 4.16 – Required E_b/N_0 for different code block sizes, given a fixed allocation of 1440 channel bits.

Figure 4.15 shows the importance of reducing any overhead, including those introduced by higher layers. This is especially the case if smaller code block sizes are to be used, as it is expected to be the case for many eMTC applications.

Considering an eMTC smart metering application based on a command-response traffic model [3] with downlink commands of 20 bytes, the overhead introduced by E-UTRAN is typically 48 bits with 24bit for CRC, 16 bits for RLC header and 8 bits for MAC header. The total code block size becomes 26 bytes or 208 bits. From Figure 4.15, the impact of adding an IPV6 header of 40 bytes has a cost of approximately 4.3dB in SNR. For smaller optimized commands of 10 bytes and 5 bytes, the impact would increase to approximately 5.5 dB and 6.7 dB respectively. For code blocks larger than 350 bits, the impact is constant and around 3dB. The results are summarized in Table 4.8.

Table 4.8 — Impact of a 40 byte overhead for different code block sizes on EPA-1 channel

RLC SDU size (bytes)	Code block size (bits)*	Impact in SNR of an additional 40 byte overhead (dB)
5	88	~6.7 dB
10	128	~5.5 dB
20	208	~4.3 dB
> 38 bytes	> 352 bits	~3 dB

* Assuming a fixed overhead of 48 bits accounting for RLC/MAC header + CRC

These results illustrate that coverage enhancement for eMTC calls for a highly efficient protocol stack with minimum overhead.

Non-IP access was partially introduced by 3GPP in release 13 [4] and can be an important coverage enhancement tool. The current 3GPP restriction in using the very low block sizes [2] may, however, limit some of the benefits of a highly optimized application.

4.5.2. Turbo coding vs convolutional coding

The discussion in the previous section enhances the non-linear relation between SNR and code block size at low spectral efficiencies. In terms of coverage enhancement, smaller code blocks are clearly preferable over larger ones. However, from a transport block point of view, having very small transport blocks has an extra negative effect in spectral efficiency due to the fixed overhead introduced by the CRC block, as well as the RLC and MAC headers [6].

A further investigation is now done regarding the channel coding performance. While turbo coding is a capacity approaching code [29], its high performance comes from the use of a large internal interleaver, which requires the use of a large code block sizes. On very smaller code block sizes, the BER/BLER performance of the turbo coder is highly degraded.

On the other hand, while the BER performance of tail-biting convolutional codes is mostly independent on the block size [32], the BLER performance degrades with increasing code block sizes. This is an intuitive result as for a given BER, larger blocks will likely have more errors than smaller ones.

In Figure 4.17, the simulated turbo coding BLER performance, at the native rate of 1/3 is compared to the 1/3 tail-biting convolutional code over AWGN channel with QPSK modulation. The used code block sizes include a 24 bit CRC for BLER evaluation. The results confirm that, as the block size decreases, the BLER performance also decreases for turbo coding, while increasing for convolutional coding.

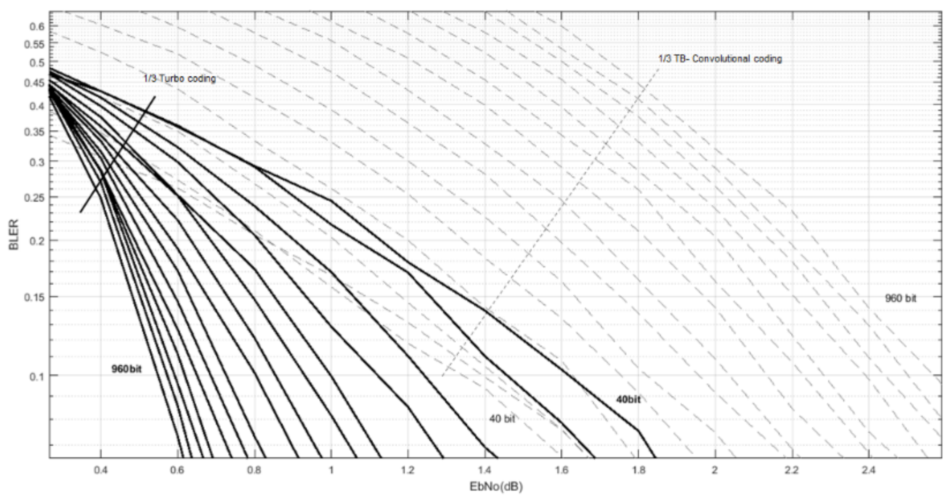


Figure 4.17 Simulation results of 1/3 Turbo coding vs 1/3 Tail-biting convolutional code for code block sizes: 40, 56, 80, 112, 144, 176, 232, 280, 352, 432, 528, 624, 736, 832, 960 bits, over AWGN.

Further treatment of these results, for the points with 10% BLER, is shown in Figure 4.18. For small code block sizes, the convolutional code approaches the performance achieved by the turbo coding, outperforming it for code block sizes approximately below 80 bits.

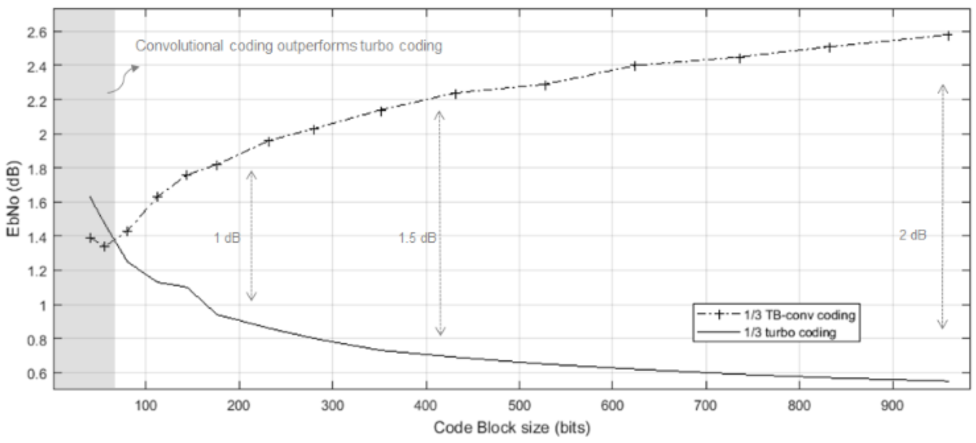


Figure 4.18 Simulation results of 1/3 Turbo coding vs 1/3 Tail-biting convolutional code for different block sizes and for a BLER of 10% over AWGN channel.

The simulation configuration in Table 4.7 is reused to simulate five small code block sizes approximately matching the use case scenarios listed in Table 4.8. The simulation is performed for tail-biting convolutional coding and the results are compared to the turbo coding figures. As shown in Figure 4.19, the performance degradation by using the convolutional code is, at most, 0.8 dB at 10% BLER for the largest block size of 232 bit.

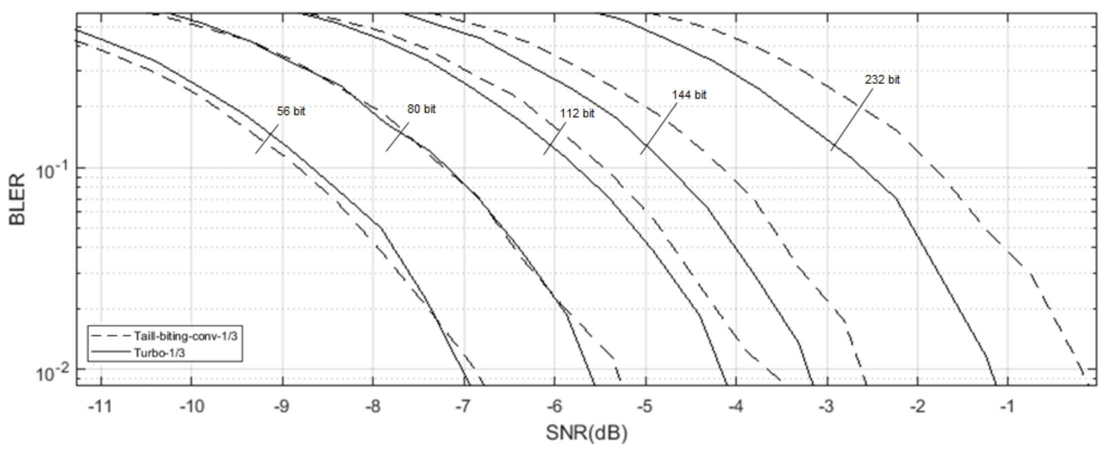


Figure 4.19 Simulation results of 1/3 Turbo coding vs 1/3 Tail-biting convolutional code for different block sizes over EPA-1 channel.

For small transport block sizes, a less power-hungry channel coding may nearly match the performance of computational intensive Turbo coding. Using convolutional coding, can further help in reducing the terminal power consumption, memory requirements and cost for specific eMTC related applications.

Chapter 5

Conclusion

The main contribution of this thesis is the development of a link-level simulation framework capable of prototyping and investigating concepts related to MTC devices.

Throughout this text, the flexibility of the framework has been illustrated in a number of examples and simulations. Section 2.2- presented the system data structure as a major contribution to the framework flexibility, both in terms of OFDM numerology configuration, as well as pilot pattern definitions. An example of a hypothetical, non-LTE configuration is given 2.2.6.

The possibilities added by the channel masking concept is explained in section 3.3- , and explorer further in the example of section 3.3.2.

The possibilities allowed by the different combining options for HARQ and repetitions are initially explored in section 3.2- . Sections 4.2 - and 4.3 - investigated these topic further on a variety of configurations, illustrating, on one hand, that the simulated results are aligned with the published results, and, on the other hand, that the configuration flexibility, allows also for non-standard configurations.

Section 4.5 - presented an investigation on the impact of code block sizes on the DL SNR performance. This was done also for sizes not allowed by the current 3GPP standards for the simulated context. The results illustrated how an efficient and light protocol stack, with minimum overhead, may be a valuable asset to enhance coverage of particular low bit rate application, like smart metering. It was shown that IPV6 header alone has an impact of approximately 3 dB in SNR for large code block sizes, and up to 6.7 dB in highly optimized applications with small code block sizes.

The investigation was further developed to show that, for small block sizes, the use of convolutional code for the DL PDSCH may be beneficial in terms of device complexity, power consumption and cost, when compared to the standardized more complex turbo codes.

5.1- Limitations and future work

While providing a rich set of feature and configuration options, it is important to understand the limitations of the current framework and areas to be considered for future improvements.

Realistic, pilot-base channel estimation is among the first candidates for future improvements. The current implementation supports perfect channel estimation only, therefore providing an upper performance bound. A channel estimation module, as described in Annex D.1.7, is already available to accommodate a future implementation. A realistic channel estimation implementation would allow a more accurate study of the real impact of a particular change.

The current implementation does not introduce any synchronization issues, modeling therefore perfectly synchronized transceivers. Among these issues, Carrier Frequency Offset (CFO), is likely the 1st implementation choice. This would allow for possibilities in studying ways to contract CFO.

The current equalization is based on a single tap and zero forcing algorithm, as detailed in D.1.8. More advanced equalization may be needed, particularly in conjunction with realistic channel estimation, and eventually fast-varying channels.

While the framework only implements the downlink, it may easily be extended for UL, by adapting the lower stages of the physical layer.

From a usability point of view, it may be beneficial to add additional return data from the framework, including throughput analysis, as well as additional print levels for debugging purposes. Inclusion of confidence intervals to the simulated points may help to interpret results across different runs. Finally, while the code was written with the goal of easy maintenance, rather than performance, it is surely possible and desirable to optimize the code to increase performance.

Annex A

This Annex lists the framework APIs accessed by the main modules, the relevant data structure, and a test script example.

A.1 - APIs accessed by the script

A.1.1. systemConfig()

Loads a predefined system configuration into the system data structure.

Prototype

```
system
systemConfig( typeConfig )
```

Parameters

typeConfig: a tag linked to a predefined system configuration. The currently pre-defined configurations are the following:

```
'LTE_normalCP-1.4Mhz_1Tx'
'LTE_normalCP-1.4Mhz_2Tx'
'LTE_normalCP-3Mhz_1Tx'
'LTE_normalCP-3Mhz_2Tx'
'LTE_normalCP-5Mhz_1Tx'
'LTE_normalCP-5Mhz_2Tx'
'LTE_normalCP-10Mhz_1Tx'
'LTE_normalCP-10Mhz_2Tx'
'LTE_normalCP-15Mhz_1Tx'
'LTE_normalCP-15Mhz_2Tx'
'LTE_normalCP-20Mhz_1Tx'
'LTE_normalCP-20Mhz_2Tx'
'LTE_extendedCP-20Mhz_1Tx'
'LTE_extendedCP-20Mhz_2Tx'
'LTE_extendedCP-5Mhz_1Tx'
'LTE_extendedCP-5Mhz_2Tx'
```

Return

system: the populated system data structure.

A.1.2. UEConfig()

Loads a predefined UE configuration into the UE data structure.

Prototype
```
UE
UEConfig( typeConfig )
```

Parameters
typeConfig: a tag linked to a predefined UE configuration. The currently pre-defined configurations are the following:

```
'cat-6_1RX'
'cat-M1_1RX'
```

Return
UE: the populated UE data structure.

A.1.3. linkConfig()

Loads a predefined link configuration into the link data structure.

Prototype
```
link
phyChannelConfig( typeConfig, CBsize);
```

Parameters
typeConfig: a tag linked to a predefined link configuration. The currently pre-defined configurations are the following:

```
'CRC24_16QAM'
'CRC24_QPSK'
'CRC24_QPSK_conv1/3'
'CRC24_16QAM_conv1/3'
```

CBsize: Optional parameter to configure the link with a specific code block size.

Return
link: the populated *link* data structure.

A.1.4. simulatorManager()

Initiates a new simulation run, or resumes a previously interrupted simulation.

Prototype
```
outcome, status
simulatorManager(run, fullpath)
```

Parameters
run: run data structure, containing all configuration instances to be simulated.

fullpath: full path to the test script calling this function. Typically set to `mfilename('fullpath')`.

Return
outcome: the *run* data structure populated with the simulation outcomes.

status: flag indicating the validity of the simulation results:

0: success
1: failure

A.2 - APIs accessed by the simulator

A.2.1. antennaMapping ()

Performs antenna mapping and precoding, depending on antenna configuration. Current configurations include SISO and 2 transmit antennas with transmit diversity, according to 36.211 section 6.3.3.3/ 6.3.4.3.

Prototype
```
out
antennaMapping(in, config)
```

Parameters
in: input vector of complex modulated symbols.

config: config data structure.

Return
out: output vector of complex modulated symbols. Extra dimension is added in MISO case.

A.2.2. awgnChannel ()

Adds Additive white Gaussian Noise to the input signal.

Prototype
```
out,noisePowerdB
awgnChannel(in, snr, config)
```

Parameters
in: input vector of complex modulated symbols.

snr: the target Signal-to-Noise Ratio

config: the config data structure.

Return
out: output vector of complex modulated symbols with added AWGN.

noisePowerdB: Noise power in dB, added by the AWGN block.

A.2.3. channelCoding()

Applies channel coding on the input data.

Prototype
```
out, codeRate
channelCoding(in, config)
```

Parameters
in: input vector of bits.

config: the config data structure.

Return
out: vector of coded bits.

codeRate: the effective code rate used.

A.2.4. channelDecoding()

Applies channel decoding on the input data.

Prototype
```
out
channelCoding(in, config)
```

Parameters
in: input vector of hard or soft-bits.

config: the config data structure.

Return
out: vector of decoded bits.

A.2.5. chEstimation()

Performs channel estimation on the input data grid.

Prototype
```
out
chEstimation(in, Hperfect, pilotSeq, config)
```

Parameters
in: matrix of complex elements, representing the received OFDM grid.

config: the config data structure.

Return
out: matrix of complex elements, representing the channel gains on the OFDM grid. A dimension is added with a size equal to the number of transmit antennas.

A.2.6. CRC_attachment()

Add a CRC tail to the input vector.

Prototype
```
out
CRC_attachment(in, config)
```

Parameters
in: vector, representing a block of bits.

config: the config data structure.

Return
out: vector, representing a block of bits. Same as input with extra CRC tail.

A.2.7. CRC_check()

Performs the CRC check on the input block for bits.

Prototype
```
Out, error
CRC_check(in, config)
```

Parameters
in: vector, representing a block of bits with CRC tail.

config: the config data structure.

Return
out: vector, representing a block of bits with the CRC removed.

error: flag indicating the outcome of the CRC check:

0: success
1: failure

A.2.8. demapping()

Retrieves the data from the OFDM grid, based on the scheduling map.

Prototype
```
rxData, rxPilot
demapping(in, config, scheduling)
```

Parameters
in: matrix of complex elements, representing the received OFDM grid.

config: the config data structure.

scheduling: matrix of complex elements, representing the scheduled resources on the OFDM grid.

Return
rxData: vector, representing a block of de-mapped data symbols.

rxPilot: vector, representing a block of de-mapped pilot symbols.

A.2.9. equalization ()

Retrieves the data from the OFDM grid, based on the scheduling map.

Prototype
```
out
equalization(in, h, SNR, config, scheduling)
```

Parameters
in: vector of complex elements.

h: channel estimation matrix covering the entire OFDM grid and for all TX antennas.

snr: Signal-to-Noise Ratio estimation

config: the config data structure.

scheduling: matrix of complex elements, representing the scheduled resources on the OFDM grid.

Return
out: vector, representing a block of equalized data symbols.

A.2.10. fadingChannel ()

Filters the input signal with a fading equivalent channel.

Prototype
```
out, Hperfect
fadingChannel(in, config)
```

Parameters
in: vector of complex elements.

config: the config data structure.

Return
out: vector of complex elements, representing the faded signal.

Hperfect: perfect channel estimation matrix covering the entire OFDM grid and for all TX antennas.

A.2.11. getNewTB ()

Retrieves a new transport block.

Prototype
```
out, config
getNewTB(config)
```

Parameters
config: the config data structure.

Return
out: a vector with a sequence of bits representing a transport channel.
config: the updated config data structure.

A.2.12. OFDM_demod_tti ()

Performs OFDM demodulation on an input signal.

Prototype
```
out
OFDM_demod_tti(in, config)
```

Parameters
in: vector of complex elements, representing the received time domain signal.

config: the config data structure.

Return
out: matrix of complex elements, representing the received OFDM grid.

A.2.13. OFDM_mod_tti ()

Performs OFDM modulation on an input signal.

Prototype
```
out
OFDM_mod_tti(in, config)
```

Parameters
in: matrix of complex elements, representing the received OFDM grid

config: the config data structure.

Return
out: vector of complex elements, representing the time domain signal.

A.2.14. QAM_demod ()

Performs QAM de-modulation of an input signal.

Prototype
```
out
QAM_demod(in, Nvar, config)
```

Parameters
in: vector of complex elements, representing the modulated signal

Nvar: estimated noise variance, representing the modulated signal

config: the config data structure.

Return
out: vector, representing the demodulated signal in bits of soft-bits.

A.2.15. QAM_mod ()

Performs QAM modulation of an input signal.

Prototype
```
out
QAM_mod(in, config)
```

Parameters
in: vector, representing the input bit signal

config: the config data structure.

Return
out: vector of complex symbols, representing the modulated signal.

A.2.16. rateMatchingRX ()

Performs the reverse rate-matching operation.

Prototype
```
out
rateMatchingRx(in, config)
```

Parameters
in: vector, representing the input bit signal

config: the config data structure.

Return
out: vector, representing the output bit signal

A.2.17. rateMatchingTX ()

Performs the rate-matching operation.

Prototype
```
out
rateMatchingTx(in, config)
```

Parameters
in: vector, representing the input bit signal

config: the config data structure.

Return
out: vector, representing the output bit signal

A.2.18. reTxCombiner ()

Performs the symbol level combining operation.

Prototype
```
out
reTxCombiner(in, h, config, scheduling, type)
```

Parameters
in: vector of complex symbols.

h: channel estimation matrix covering the entire OFDM grid and for all TX antennas.

config: the config data structure.

scheduling: matrix of complex elements, representing the scheduled resources on the OFDM grid.

type: buffer operation, combine or reset:

```
'updateBuffer'
'initBuffer'
```

Return
out: vector of complex symbols, representing the result of the combining operation.

A.2.19. scheduler ()

Performs the scheduling of OFDM resources among all registered channels.

Prototype
```
scheduling, grid, config
scheduler(config, rbn, h)
```

Parameters
config: the config data structure.

rbn: an integer containing the current RBn.

h: channel estimation matrix covering the entire OFDM grid and for all TX antennas.

Return
scheduling: matrix of complex elements, representing the scheduled data resources on the OFDM grid.

grid: the grid data structure, containing the allocated resources for the selected channels and pilots. All not allocated resources are free for data scheduling.

config: the updated config data structure.

A.2.20. snrCompensation ()

Calculated the relation SNR/EbNo. Used when the subInstance is defined in EbNo and AWGN is added based on SNR.

Prototype

```
out
snrCompensation(config, codeRate)
```

Parameters

config: the config data structure.

codeRata: the effective codeRate at the channel coding stage.

Return

out: the amount of SNR compensation needed, in dB, to scale SNR from EbNo.

A.2.21. symbolMapping ()

For each of the configured channels and pilots, performs the mapping to the allocated OFDM resources.

Prototype

```
out, pilotSeq
symbolMapping(in, config, grid, scheduling)
```

Parameters

in: vector of complex elements. It has 2 dimensions in case of multiple transmit antennas.

config: the config data structure.

grid: the grid data structure, containing the allocated resources for the selected channels and pilots. All not allocated resources are free for data scheduling.

scheduling: matrix of complex elements, representing the scheduled data resources on the OFDM grid.

Return

out: matrix of complex elements, representing the mapped symbols into the OFDM grid. There is one grid per antenna port, therefore the matrix may have 2 or 3 dimensions.

pilotSeq: vector of complex elements containing the pilot sequence used.

A.2.22. updateTiming ()

Updates the counters associated with the system timing.

Prototype

```
rbnNext, sfnNext
updateTiming(config, rbnLast)
```

Parameters

config: the config data structure.

rbnLast: the previous RBn value.

Return
rbnNext: the next RBn value.

sfnNext: the next SFN value.

A.3 - APIs accessed by the simulation manager

A.3.1. simulator ()

Updates the counters associated with the system timing.

Prototype
```
results, stats
simulator(config, subInstance)
```

Parameters
config: the config data structure.

subInstance: integer identifying the intended subInstance of the configuration to be run.

Return
Results: structure containing the simulation results

stats: structure containing additional statistics on the simulation run.

A.4 - Data structures

A.4.1. system

```
system.BW                              : system total BW (MHz)
system.SCspacing                       : SC spacing (Hz)
system.nTx                             : Number Tx antennas
system.timing.sfn                      : SFN duration (in #RB duration)
system.timing.sfnMax                   : SFN maximum before wrapping
system.nRB                             : effective BW (in #RB BW)
system.NFFT                            : NFFT size used
system.RB.nSC                          : #SC in a Resource Block
system.RB.nsymbol                      : #OFDM symbols in a Resource Block
system.cpNSamples(system.RB.nsymbol)   : CP length of each OFDM symbol in the RB
system.TsOFDM                          : OFDM symbol duration (seconds)
system.Ts                              : base time unit (seconds)
system.Fs                              : sampling freq (Hz)
system.nSC                             : effective BW (in #SC)
system.pilot.idx(system.nTx)           : pilot grid on each Tx antenna
system.pilot.PowerBoost                : pilot power boost
system.channels{n}.id                  : channel id/name
system.channels{n}.idx                 : channel grid mapping
system.channels{n}.start               : timming of 1st transmisson (in #RB)
system.channels{n}.RBrep               : repetition interval (in #RB)
```

A.4.2. link

link.	: CRC size/type
link.ModOrder	: Modulation order
link.Qm	: bits/symbol
link.rv	: redundancy version
link.CBsize	: codeblock size
link.tti	: TTI duration in #RB duration
link.G	: channel size in # channel bits for one tti
link.coding	: channel coding to be used

A.4.3. UE

UE_capability.Nsoft	: Total # soft bits.
UE_capability.Mdl_harq	: # of Harq processes. Not yet used
UE_capability.nRx	: # receive antennas. Not yet used

A.4.4. config

config.system	: the system data structure
config.UE	: the UE data structure
config.link	: the link data structure
config.enableCRC	: enables/disables CRC related blocks
config.enableCoding	: enables/disables coding related blocks
config.enableModulation	: enables/disables modulation related blocks
config.enableSoftDemodOut	: enables/disables soft-bit demodulation
config.enableRateMatching	: enables/disables rateMatching blocks
config.enableHARQ	: enables/disables HARQ functionality
config.enableRepetitions	: enables/disables repetition functionality
config.enableFixedCodeRate	: enables/disables fix code rate allocation
config.enableOFDM	: enables/disables OFDM related functions
config.enableAwgn	: enables/disables AWGN channel
config.enableFading	: enables/disables fading channel
config.typeFading	: selects fading channel, if enabled
config.maxRepetitions	: maximum number of repetitions (if enabled)
config.RVperiod	: consecutive repetitions with same RV
config.maxHarqRetx	: maximum number of HARQ reTx (if enabled)
config.forceHarqChaseComb	: disables RV change for HARQ/repetitions
config.retxCombineMethod	: repetition combining method
config.schedulingType	: scheduling algorithm
config.scheduledNumRB	: scheduling size in RBs
config.scheduledRBStart	: scheduling start (localized case)
config.coderateTarget	: target code rate if fixed code rate is enabled
config.txAntannaCorrelation	: antenna correlation
config.chEstimationType	: type of channel estimation to be used
config.equalizationType	: type of equalization to be used
config.sim.ebnoRange	: vector with all subInstances (EbNo in dB)
config.sim.targetNumErrors	: target number of bit errors
config.sim.maxTxBits	: target number of transmitted bits
config.sim.targetTxTTI	: target number of TTIs (total transmission time)
config.sim.BLERtarget	: target BLER
config.fadingRandomSeed	: fading random seed
config.printLevel	: print level to the console during simulation

A.4.5. run

```
run{}.config                           : config struct
run{}.results.berRange()               : ber results for all subInstances
run{}.results.blerRange()              : bler results for all subInstances
run{}.results.snrRange()               : subInstances in terms of snr
run{}.results.ebnoRange()              : subInstances in terms of ebno
run{}.completedRangeIdx                : last completed subInstance
run{}.status                           : simulation status (started/terminated)
run{}.timeLastRun                      : datetime of last run
run{}.singlPointSimTime                : simulation time for last subInstance
```

A.4.6. status

```
status.script                          : script name
status.state                           : simulation state
status.path                            : script full path
```

A.5 - Test script example

Script example for the simulation configuration Table 4.7 with the outcome shown in Figure 4.14.

```
%% ********************************************************************
%
%   script to investigate BLER vs SNR/EbNo with different CB sizes
%
%
%% ********************************************************************
clear all;

%% initialize global config struct with relevant simulation data
config.system                 = systemConfig('LTE_normalCP-10Mhz_2Tx');
config.UE                     = UEConfig('cat-6_1RX');
config.link                   = phyChannelConfig('CRC24_QPSK', 0);

config.enableCRC              = true;
config.enableCoding           = true;
config.enableModulation       = true;
config.enableSoftDemodOut     = true;
config.enableRateMatching     = true;
config.enableHARQ             = false;
config.enableRepetitions      = false;
config.enableFixedCodeRate    = false;
config.enableOFDM             = true;

config.enableAwgn             = true;
config.enableFading           = true;
config.typeFading             = 'EPA-1';

config.schedulingType         = 'fixed-localized';
config.scheduledNumRB         = 6;
config.scheduledRBStart       = 2;
config.txAntannaCorrelation   = 'low';
config.chEstimationType       = 'perfect';
config.equalizationType       = 'zf';
config.CombinePreEq           = 0;

config.sim.ebnoRange          = 0:0.2:8;
config.sim.targetNumErrors    = 1e8;
config.sim.maxTxBits          = 1e8;
config.sim.targetTxTTI        = 100;
config.sim.BLERtarget         = 0.01;
config.fadingRandomSeed       = 124;
config.printLevel             = 2;
```

```
%% other initializations
CBsize = [40 56 80 112 144 176 232 280 352 432 528 624 736 112 960];
endRun = length(CBsize);
run = cell(1,endRun);

%% configure the simulation runs
for instance = 1:endRun

    % update CBsize for next run
    config.link.CBsize = CBsize(instance);
    run{instance}.config = config;
end

%% run the simulator
[outcome, status] = simulatorManager(run, mfilename('fullpath'));
if ~status
    fprintf('\nError running the simulator.\n');
    return
end

%% plot the results
figure
for instance = 1:endRun
    instResults = outcome{instance}.results;
    instConfig = outcome{instance}.config;
    semilogy(instResults.snrRange, instResults.blerRange, ’black’);
    hold on
end
grid on;
title('BLER vs SNR for different code block sizes(EPA-1)');
xlabel('SNR(dB)');
ylabel('BLER');
```

Annex B

This Annex presents framework profiling data, identifying the key bottlenecks modules in terms of simulation performance.

B.1 - Performance overview

The following performance figures were obtained using the MATLAB profiling tools. While Turbo Decoding is clearly the bottleneck on AWGN environments, when fading is used, the fading channel (including the perfect channel estimation function) accounts for nearly half of the total processing time.

Table B.1 — Top 3 processing modules with highest processing time, AWGN

Parameter	% processing time
Turbo Decoding	24%
OFDM encoder	12%
Scheduler	8%

Table B.2 — Top 3 processing modules with highest processing time, fading

Parameter	% processing time
Fading Channel	46%
Turbo Decoding	6%
OFDM encoder	3%

Annex C

This Annex presents the MATLAB System Objects used from the Communications System toolbox.

C.1 - MATLAB Communications System Objects uses

Table C.3 — MATLAB System Objects used

Function	MATLAB system object
AWGN channel	comm.AWGNChannel* [53]
Fading channel	comm.MIMOChannel* [54]
CRC attachment	comm.CRCGenerator* [49]
1/3 Turbo Coding	comm.TurboEncoder* [48]
1/3 tail-biting convolutional code	comm.ConvolutionalEncoder* [51]
Modulation	comm.RectangularQAMModulator* [46]
CRC check	comm.CRCDetector* [50]
1/3 Turbo decoding	comm.TurboDecoder* [47]
1/3 tail-biting convolutional code	comm.ViterbiDecoder* [52]
Demodulation	comm.RectangularQAMDemodulator* [45]

* From the MATLAB Communications System toolbox

Annex D

This Annex presents implementation details for the relevant blocks accessed by the simulator, and that were not included in section 3.1- .

D.1 - Functional Blocks not described in 3.1-

D.1.1. The CRC attachment and CRC check blocks

The CRC attachment block adds a CRC of size m to a block of size n. This is the first user plane processing step within the physical layer. In the framework this is implemented on the *CRC_attachment.m* module using the MATLAB *comm.CRCGenerator* system object.

The behavior of this module is controlled by two input parameters as shown in Figure D.1. The *config.link.CRCtype* allows the selection of a predefined cyclic generator polynomials identified by a tag. Currently the module implements the LTE standardized polynomials [1] for 8bit, 16bit, and the two 24 bit versions. The second parameter, *config.enableCRC*, allows bypassing the CRC operation, so that no CRC is attached.

The CRC check module, receives a block of size n+m and performs a CRC check based on the last m bits of the block. A Boolean output returns the CRC check result. The implementation is based on the MATLAB comm.CRCDetector and the configuration is, in all aspects the same as for the CRC attachment case. This module may be disabled with the Boolean parameter *config.enableCRC*, so that no error check is performed, and the output becomes the same as the input.

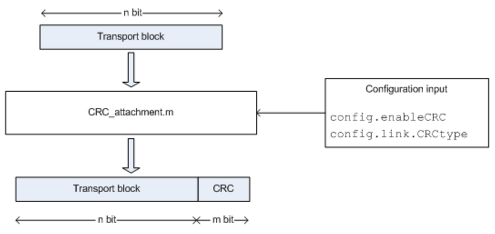


Figure D.1 – CRC attachment block.

D.1.2. The Code block segmentation and concatenation blocks

Code block segmentation is included in the LTE specification for block sizes above 6144 bit. With eMTC devices in mind, where block sizes are expected to smaller than 1000 bit, code block segmentation, as standardized in [1] is not expected to ever be used. The framework therefore does not implement code block segmentation.

D.1.3. The modulator and demodulator blocks

The modulator performs the constellation mapping of the incoming bit sequence of size *n*, according to the selected modulation. The output is a block of complex modulated symbols as illustrated in Figure D.2. The framework currently supports the 3GPP standardized modulations for category M1 devices, QPSK and 16QAM. The implementation is based on MATLAB's *comm.RectangularQAMModulator* system object, configured to perform the 3GPP symbol mapping as per [11].

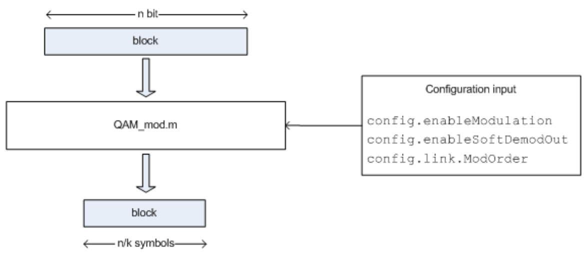


Figure D.2 – QAM Modulator

In the standard configuration, the demodulator receives a sequence of symbols and performs the demodulation operation, providing at the output a sequence of soft-bit in the form of log-likelihood ratio (LLR) [44], as shown in Figure D.3. The number of output symbols is given by n/k, where k is the number of bits per symbol, computed from *config.link.ModOrder*.

It is possible, however, to enforce hard-bit detection by setting the *config.enableSoftDemodOut* to true. The implementation is based on MATLAB's *comm.RectangularQAMDemodulator* system object.

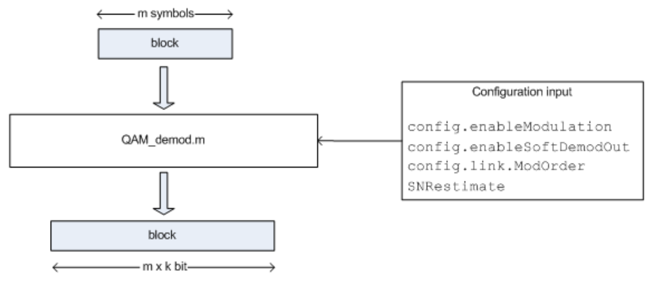


Figure D.3 – QAM Demodulator

It is possible to bypass the modulation and demodulation operation with the setting *config.enableModulation*.

D.1.4. The Antenna mapping module

The framework supports several transmit antennas from a system configuration point of view. However, the only multi-antenna scheme currently implemented is transmit diversity with two antennas, using a variant of the Alamouti scheme [56], the Space Frequency-Block Coding (SFBC) approach, as defined for LTE in [11]. Implementation is based on the generic two-step multi-antenna processing stage defined for LTE. The incoming block of symbols are first mapped to two different layers, corresponding to the two antenna ports. The mapping can be seen as a simple serial to parallel conversion, where all incoming even and odd symbols are mapped to the layers corresponding to antenna port0 and antenna port1, respectively. The second step is the precoding phase in which each layer is mapped to the actual antenna port by the following precoding operation:

$$\begin{bmatrix} y_{2i}^{(0)} \\ y_{2i}^{(1)} \\ y_{2i+1}^{(0)} \\ y_{2i+1}^{(1)} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & j & 0 \\ 0 & -1 & 0 & j \\ 0 & 1 & 0 & j \\ 1 & 0 & -j & 0 \end{bmatrix} \begin{bmatrix} \mathrm{Re}(x_i^{(0)}) \\ \mathrm{Re}(x_i^{(1)}) \\ \mathrm{Im}(x_i^{(0)}) \\ \mathrm{Im}(x_i^{(1)}) \end{bmatrix}, \tag{D.1}$$

Where $x_i^{(p)}$ represent the i^{th} symbol on layer *p*, and $y_j^{(p)}$ represent the jth symbol on antenna port p, with p $\in$ {0,1}.

The overall process is effectively dividing the power among the two antennas, replicating the input symbols into antenna port0, while operating on consecutive pair of symbols for antenna port1, as shown in Figure D.4.

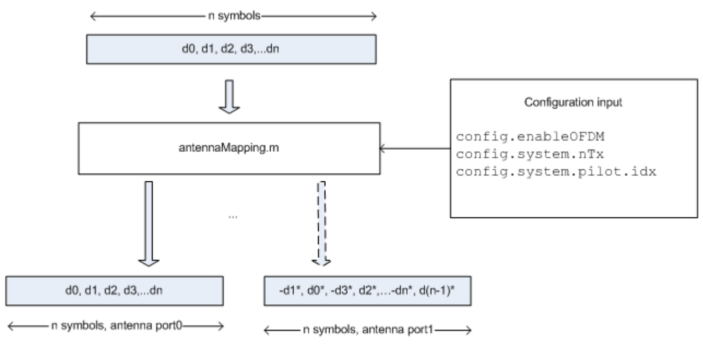


Figure D.4 – Antenna mapper

Since transmit diversity relies on the assumption that the channel is unchanged between the pair of symbols sent, some care must be taken if attempting to use the current transmit diversity implementation for other non-LTE OFDM configurations, especially due to pilot configuration. The LTE pilot configuration ensures that two data symbols are always adjacent in the frequency domain, with no pilot in between.

Adding support for additional multi-antenna schemes requires implementation of the corresponding layer mapping and precoding operations. An additional field to the *config* structure is required to identify the scheme to be used.

D.1.5. The fading channel block

The modeling of multi-path fading propagation is a crucial component of any wireless simulation environment. The framework relies on a statistical multi-path channel model implemented by the MATLAB *comm.MIMOChannel* system object. This system object is a generic implementation of Multiple-Input Multiple-Output (MIMO) multipath fading channel, supporting both Rayleigh and Rician fading, Doppler spectrum and maximum Doppler shift configuration, as well as antenna correlation definition [54]. The different paths are modeled using a delay profile in the form of a Tapped Delay-Line (TDL), containing the relative path delays and gains of the resolvable paths.

LTE defines a set of channel models, representing low, medium and high delay spread environments [13]. Each channel model may be associated with a particular maximum Doppler frequency. The framework supports all these channel models at various Doppler shifts, selectable in *config.typeFading*.

The framework currently supports only one receive antenna, in line with 3gpp CAT-M1 devices. The *comm.MIMOChannel* system object is, therefore, defined either as Multi-Input Single-Output (MISO), or Single-Input Single-Output (SISO), depending on the number of configured transmit antennas. In case of more than one transmit antenna, the amount of antenna correlation is defined by a correlation matrix for low, medium and high correlation, as specified in [13]. These are configurable with *config.txAntennaCorrelation*.

An overview of the fading channel block is shown in Figure D.5. The module will receive one transmission block per active antenna, as configured though the *config.system*.nTx field. The module will then instantiate the *comm.MIMOChannel* system object in order to apply the selected fading model, outputting one single filtered block with the same size as the input blocks.

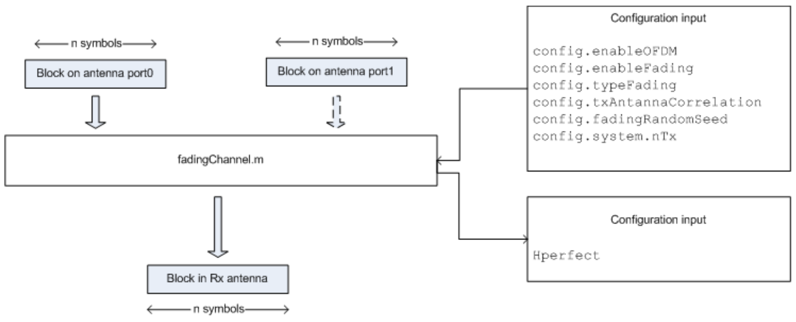


Figure D.5 – Fading channel

A second output from the *fadingChannel.m* module is the perfect frequency based channel estimation. The channel estimate is provided for each element in the OFDM grid, and for each of the transmit antennas. The calculation is based on the channel path gains returned by the *comm.MIMOChannel* system object. For each time-domain sample i, and for each transmit antenna, the *comm.MIMOChannel* system object returns an n element row vector containing the channel gains for each of the n channel taps configured. The samples corresponding to the cyclic prefix are discarded, and all vectors, corresponding to *NFFT* consecutive samples, are averaged, obtaining the mean channel gains over each of the k OFDM symbols. In order to obtain the time domain channel response, the *n* average tap gains are time scaled according to their corresponding delay profile configuration. However, the sampling rate resolution is too low for accurate magnitude frequency response estimation, especially at low values of *NFTT*. Therefore an oversampling factor of 20 is used (100 for 128 point FFT cases). The time-scaled version of is then converted to the frequency domain by means of the FFT of size *k.NFFT*, where k is the oversampling factor. In the frequency domain the signal is down sampled and filtered to the matching OFDM grid bandwidth size.

The fading module may be disabled either by the *config.enableOFDM* or *config.enableFading*. If fading is disable, all the incoming blocks, corresponding to different antenna ports are simply added to generate the output block. It is also possible to configure an initial random seed, *config.FadingRansomSeed*.

D.1.6. The AWGN channel module

The AWGN channel module is implemented in *awgnChannel.m*, based on MATLAB's *comm.AWGNChannel* system object.

The module adds Additive White Gaussian Noise (AWGN) to the complex input sequence based on the parameted *targetSNR*. The module calculates and returns the added noise power in *noisePowerdB*.

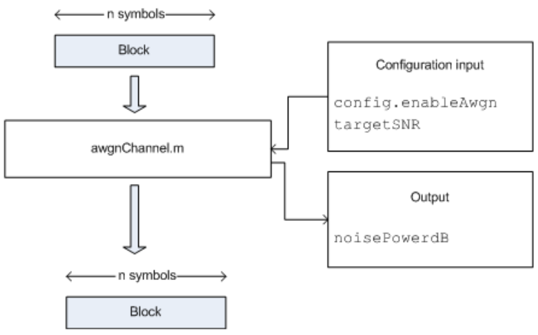


Figure D.6 – AWGN channel

D.1.7. Channel estimation

Channel estimation is a major function within the receiver structure, with direct impact of the overall performance. Currently, the framework only supports perfect channel estimation, calculated in *fadingChannel.m* as described in D.1.5. This can be seen as providing an upper bound on performance. Also, realistic pilot-based channel estimation, is tightly connected to the actual pilot layout, and therefore difficult to generalize [30].

The module *chEstimation.m* is a placeholder for realistic pilot-based channel estimation implementations. The module is to perform the estimation on the received OFDM grid based on the pilot sequence, pilot positions, and number of antennas ports, given as arguments. The current output is a copy of the input matrix *Hperfect*.

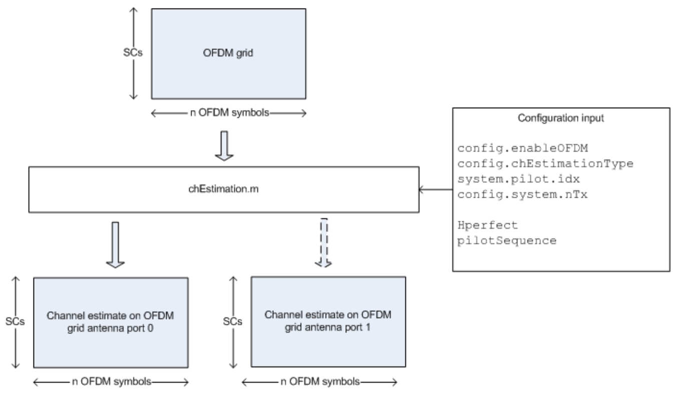


Figure D.7 – Channel estimation channel

D.1.8. Equalization block

The channel equalization receives the de-mapped symbols and performs equalization based on the channel estimation, received as input arguments, and the selected algorithm. Currently only the one tap, Zero Forcing (ZF) algorithm is implemented [30], however, *SNRestimation* is provided as an argument for more advanced implementations, like the Minimum Mean Squared Error (MMSE) equalizer.

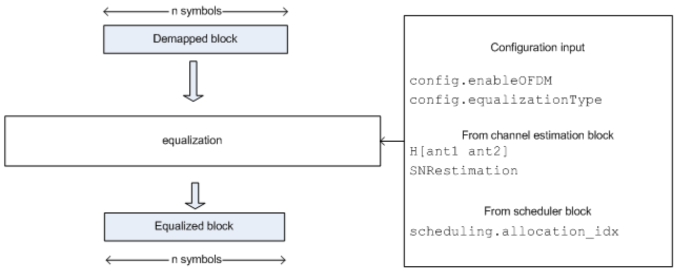


Figure D.8 – Channel equalization

In case of transmit diversity scenarios, the classical linear combining method as presented in [55] is used, adapted for SFBC, and with channel estimation on each antenna averaged over two consecutive data subcarriers.

The equalizer block is disabled if *config.enableOFDM* is set to *false*.

References

[1] 3GPP TS 36.212: "Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding".

[2] 3GPP TS 36.213: "Physical layer procedures". v13.3.0.

[3] 3GPP TR 36.888: " Study on provision of low-cost Machine-Type Communications (MTC) User Equipments (UEs) based on LTE ". v12.0.0.

[4] 3GPP TR 23.720 “Study on architecture enhancements for Cellular Internet of Things”. v13.3.0

[5] 3GPP TR 23.720 “Study on architecture enhancements for Cellular Internet of Things”. v13.3.0

[6] 3GPP R1-155034 Ericsson, “PUSCH transmission for MTC”

[7] 3GPP R1-154845 Sony, “Summary of Simulation Results for M-PDCCH”

[8] 3GPP R1-154211 Sony, “Cross PRB Channel Estimation for M-PDCCH”

[9] 3GPP R1-151214 Ericsson, “PDSCH link performance for MTC”

[10] 3GPP TS 36.101, “User Equipment (UE) radio transmission and reception”

[11] 3GPP TS 36.211, “Physical channels and modulation”

[12] 3GPP TS 36.331, “Radio Resource Control (RRC); Protocol specification”, v13.3.0

[13] 3GPP TS 36.101, “User Equipment (UE) radio transmission and reception”, v13.5.0

[14] 3GPP R1-157179, Sierra Wireless, “PUSCH RV Cycle Performance and Discussion”

[15] 3GPP R1-151588, Samsung, “Incremental Redundancy vs. Chase Combining for PDSCH Transmissions”

[16] 3GPP R1-151216, Ericsson, “PUSCH channel estimation aspects for MTC”

[17] 3GPP TR 38.912 "Study on New Radio (NR) access technology", V14.0.0

[18] ETSI GSM 03.60 “General Packet Radio Service (GPRS); Service description”, v7.4.1

[19] ETSI GSM 03.10 “GSM PLMN Connection Types”, version 3.3.0

[20]ETSI GSM 04.22 "Radio Link Protocol (RLP) for data and telematic services on the Mobile Station - Base Station System (MS-BSS) interface and the Base Station System - Mobile Services Switching Centre (BSS-MSC) interface"

[21] 3GPP 25.212 "Multiplexing and channel coding (FDD), (Release 1999)", V3.11.0

[22] 3GPP 25.212 "Multiplexing and channel coding (FDD), (Release 1999)", V5.10.0

[23] 3GPP TR 38.900 "Study on channel model for frequency spectrum above 6 GHz", v14.3.0

[24] 3GPP TR 38.804 "Study on New Radio Access Technology; Radio Interface Protocol Aspects (Release 14)", V14.0.0

[25] Stefania Sesia, Issam Toufik, Matthew Baker, “LTE, The UMTS Long Term Evolution, from theory to practice”, Wiley, 2nd Edition, 2011

[26] Bernard Sklar, “Digital Communications, Fundamentals and Applications”, Prentice Hall, 2nd Edition, 2001
[27] Erik Dahlman , Stefan Parkvall, Johan Sköld, “4G LTE-Advanced Pro and the Road to 5G”, Academic press, 3rd Edition, 2016
[28] Jung-Fu Cheng, Havish Koorapaty, “Error Detection Reliability of LTE CRC Coding”, 2008 IEEE 68th Vehicular Technology Conference
[29] C. Berrou, A. Glavieux and P. Thitimajshima, “Near Shannon Limit Error Correcting Coding and Decoding: Turbo Codes", in IEEE International Conference on Coniniunications (ICC'93), vol. 213, pp. 1064-1071, May 1993
[30] Tzi-Dar Chiueh, Pei-Yun Tsai, “OFDM Baseband Receiver Design for Wireless Communications”, John Willey and Sons, 2007
[31] A. Nimbalker, Y. Blankenship, B.Classon, T. Blankenship “ARP and QPP Interleavers for LTE Turbo Coding”, Wireless Communications and Networking Conference, 2008. WCNC 2008. IEEE
[32] Yi-Pin Eric Wang, R. Ramésh, “To Bite or Not to Bite - A Study of Tail Bits versus Tail-Biting”, Personal, Indoor and Mobile Radio Communications, 1996. PIMRC'96., Seventh IEEE International Symposium
[33] Howard H.Ma, Jack K.Wolf, “On Tail Biting Convolutional Codes”, IEEE Transactions on communications, Vol. com-34, No. 2, February 1986
[34] https://www.mathworks.com/help/comm/examples/tail-biting-convolutional-coding.html
[35] Tse.D, Viswanath.P “Fundamentals of wireless communication”, Cambridge University press, 2005
[36] Josep Colom Ikuno, Martin Wrulich, Markus Rupp, “Performance and Modeling of LTE H-ARQ”, https://publik.tuwien.ac.at/files/PubDat_173876.pdf
[37] Ericsson, “Performance Comparison of Chase Combining and Incremental Redundancy for HSDPA”, TSG-RAN Working Group 1 meeting #17
[38] David Chase, “Code Combining-A Maximum-Likelihood Decoding
[39] Approach for Combining an Arbitrary Number of Noisy packets”, IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. COM-33, NO. 5, MAY 1985
[40] R1-01-0472 Texas Instruments “Bit level and symbol level Chase combining”, TSG-RAN Working Group 1 meeting #20
[41] “Coverage Analysis of LTE-M Category-M1”, white paper. https://altair-semi.com/wp-content/uploads/2017/02/Coverage-Analysis-of-LTE-CAT-M1-White-Paper.pdf
[42] Proakis, John G, Salehi, Masoud, “Digital Communications”, McGraw-Hill, 5th edition
[43] M. Okada, S. Komaki: Pre-DFT Combining Space Diversity Assisted COFDM, IEEE Transactions on Vehicular Technology, Volume 50, Issue 2, March 2001
[44] MATLAB, Exact LLR Algorithm, https://www.mathworks.com/help/comm/ug/digital-modulation.html#brc6yjx
[45] MATLAB, comm.RectangularQAMDemodulator System object, https://www.mathworks.com/help/comm/ref/comm.rectangularqamdemodulator-class.html
[46] MATLAB, comm.RectangularQAMModulator System object, https://www.mathworks.com/help/comm/ref/comm.rectangularqammodulator-c lass.html

[47] MATLAB, comm.TurboDecoder System object, https://www.mathworks.com/help/comm/ref/comm.turbodecoder-class.html
[48] MATLAB, comm.TurboEncoder System object, https://www.mathworks.com/help/comm/ref/comm.turboencoder-class.html
[49] MATLAB, comm.CRCGenerator System object, https://www.mathworks.com/help/comm/ref/comm.crcgenerator-class.html
[50] MATLAB, comm.CRCDetector System object, https://www.mathworks.com/help/comm/ref/comm.crcdetector-class.html
[51] MATLAB, comm.ConvolutionalEncoder System object, https://www.mathworks.com/help/comm/ref/comm.convolutionalencoder-class.html
[52] MATLAB, comm.ViterbiDecoder System object, https://www.mathworks.com/help/comm/ref/comm.viterbidecoder-class.html
[53] MATLAB, comm.AWGNChannel System object, https://www.mathworks.com/help/comm/ref/comm.awgnchannel-class.html
[54] MATLAB, comm.MIMOChannel System object, https://www.mathworks.com/help/comm/ref/comm.mimochannel-class.html
[55] Goldsmith.A, “Wireless Communications”, Cambridge University Press, 2005
[56] Siavash M. Alamouti , “A Simple Transmit Diversity Technique for Wireless Communications”, IEEE Journal on selected areas in communications, Vol. 16, No. 8, October 1998
[57] Qualcomm, “Accelerating 5G NR for Enhanced Mobile Broadband”, https://www.google.se/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0ahUKEwih_pvQ59PUAhXKa5oKHRHSBq4QFgg3MAI&url=https%3A%2F%2Fwww.qualcomm.com%2Finvention%2Ftechnologies%2F5g-nr&usg=AFQjCNEhsxBoB9jIjygC9IsO8i7QRM57uw
[58] Ericsson, " Internet of Things forecast”, https://www.ericsson.com/en/mobility-report/internet-of-things-forecast
[59] Institute of TelecommunicationsVienna University of Technology, “Vienna LTE Simulators LTE-A Link Level Simulator Documentation, v1.4Q2-2016” https://owncloud.nt.tuwien.ac.at/index.php/s/2Xh3pF5ltjYCnI6#pdfviewer
[60] https://www.nt.tuwien.ac.at/research/mobile-communications/vccs/vienna-lte-a-simulators/
[61] LTE-PHY Lab https://www.is-wireless.com/5g-toolset/lte-phy-lab/
[62] MATLAB LTE System Toolbox, https://www.mathworks.com/products/lte-system.html
[63] Piro, G., Grieco, L.A., Boggia, G., Capozzi, F., Camarda, P., “Simulating LTE Cellular Systems: an Open Source Framework”, IEEE Transactions on Vehicular Technology, Vol. 60, No. 2, Feb. 2011
[64] M.Kima1, C.Gussen1, B.Espindola, T.Ferreira, W.Martins, P.Diniz, "OPEN-SOURCE PHYSICAL-LAYER SIMULATOR FOR LTE SYSTEMS", Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on Acoustics, March 2012
[65] R1-151662, Panasonic, “Evaluation on PDSCH repetition in CE MTC”