## Functions and tools

### Structures

#### Main structures

The main structures have been defined in kernel/definitions:

* The class structure **class\_t** of each dynamic library is defined in class.h

typedef struct \_class {

classid\_t id;

char \*name;

implem\_t implem;

model\_t \*model;

methods\_t \*methods;

array\_t nodearchs;

array\_t mediums;

array\_t environments;

int (\*init) (call\_t \*to, void \*params);

int (\*destroy) (call\_t \*to);

int (\*bootstrap) (call\_t \*to);

int (\*bind) (call\_t \*to, void \*params);

int (\*unbind) (call\_t \*to);

int (\*ioctl) (call\_t \*to, int option, void \*in, void \*\*out);

#ifdef \_\_cplusplus

void \*\*private\_values;

#else //\_\_cplusplus

void \*\*private;

#endif //\_\_cplusplus

} class\_t;

* The node architecture attributes **nodearch\_t** defined in nodearch.h. A node architecture is composed of:

typedef struct \_nodearch {

nodearchid\_t id;

char \*name;

char \*birth;

node\_type\_array\_t types;

array\_t classes;

array\_t transceivers;// One or several transceiver models

array\_t macs; //One or several MAC models

array\_t routings; //One or several routing models

array\_t applications; //One or several application models

array\_t interfaces; //One or several interface models

array\_t phys;

array\_t errors;

array\_t codings;

array\_t interferences;

array\_t signal\_trackers;

array\_t modulators;

array\_t \*up;

array\_t \*down;

array\_t sensors; //One or several sensors models

array\_t monitors; //One or several monitors models

classid\_t energy; //One energy model

classid\_t mobility; //One mobility model

} nodearch\_t;

* Each node in the simulation is managed through the structure **node\_t** defined in node.h

typedef struct \_node {

nodeid\_t id;

nodearchid\_t nodearch;

uint64\_t birth;

int state;

position\_t position;

#ifdef \_\_cplusplus

void \*\*private\_vars;

#else //\_\_cplusplus

void \*\*private;

#endif //\_\_cplusplus void \*noises;

node\_type\_array\_t types;

array\_t groups;

} node\_t;

* The medium structure **medium\_t** of the simulation is defined in medium.h

typedef struct \_medium {

mediumid\_t id;

char \*name;

double propagation\_range;

double speed\_of\_light; // The speed of light in meters per nanoseconds

array\_t classes;

classid\_t spectrum;

classid\_t pathloss; //One pathloss model

classid\_t shadowing; //One shadowing model

classid\_t fading; //One fading model

classid\_t interferences; //One interferences model

classid\_t intermodulation; //One intermodulation model

classid\_t noise; //One noise model

array\_t links;

array\_t modulations;

array\_t monitors;

#ifdef \_\_cplusplus

void \*\*private\_vars;

#else //\_\_cplusplus

void \*\*private;

#endif //\_\_cplusplus

} medium\_t;

* The physical environment structure **environment\_t** of the simulation is defined in environment.h

typedef struct \_environment {

environmentid\_t id;

char \*name;

array\_t classes;

classid\_t map;

array\_t physicals;

array\_t monitors;

#ifdef \_\_cplusplus

void \*\*private\_vars;

#else //\_\_cplusplus

void \*\*private;

#endif //\_\_cplusplus

} environment\_t;

#### Data structures

Several useful types have been defined in kernel/definitions/types.h file:

* **Array\_t**: An array of integers containing its size

typedef struct \_array {

int size; /\*\*< array size \*\*/

int \*elts; /\*\*< array elements \*\*/

} array\_t;

* **call\_t**: A parameter that identifies who we are calling or who has called us

#ifdef \_\_cplusplus

// The var name "class" cannot exist on the c++ world.

typedef struct \_call {

classid\_t classid; /\*\*< the called class id \*\*/

objectid\_t object; /\*\*< the called object id (node, medium, or environment) \*\*/

} call\_t;

#else //\_\_cplusplus

typedef struct \_call {

classid\_t class; /\*\*< the called class id \*\*/

objectid\_t object; /\*\*< the called object id (node, medium, or environment) \*\*/

} call\_t;

#endif //\_\_cplusplus;

* **position\_t :** A position in the 3D space

typedef struct \_position {

double x; /\*\*< x position \*\*/

double y; /\*\*< y position \*\*/

double z; /\*\*< z position \*\*/

} position\_t;

* **angle\_t:** An angle in the 3D space

typedef struct \_angle {

double xy; /\*\*< angle on the xy plane \*\*/

double z; /\*\*< angle between the xy plane and the z axis\*\*/

} angle\_t;

* **destination\_t:** A packet destination. May be a node address or a geographical position

typedef struct \_destination {

nodeid\_t id; /\*\*< the destination node id \*\*/

position\_t position; /\*\*< the destination position \*\*/

} destination\_t;

* **param\_t:** A parameter for the "init" and "bind" class functions.

typedef struct \_param {

char \*key; /\*\*< the parameter key \*\*/

char \*value; /\*\*< the parameter value \*\*/

} param\_t;

* **IntervalTree:** It is a class that represents an interval tree, i.e. it contains intervals and allows users to insert intervals, delete intervals and find all intervals that intersect a given interval. The current implementation of the IntervalTree is done with a modified RedBlackTree.

/\*\* \brief The Abstract Base Class : IntervalTree Class

\*

\* \fn Delete - deletes an interval from the tree

\* \fn Insert - inserts an interval in the tree

\* \fn GetSize - return the current size of the interval tree

\* \fn FindAllIntersections - return the intervals that intersect the given interval

\*\*/

class IntervalTree {

public:

IntervalTree(){ size\_ = 0;};

virtual ~IntervalTree(){};

void Delete(std::weak\_ptr<Interval> interval) {DeleteImpl(interval);};

void Insert(std::weak\_ptr<Interval> interval) {InsertImpl(interval);};

uint GetSize() {return size\_;};

std::list<std::weak\_ptr<Interval>> FindAllIntersections(std::weak\_ptr<Interval> interval) {return FindAllIntersectionsImpl(interval);} ;

private:

virtual void DeleteImpl(std::weak\_ptr<Interval>) = 0;

virtual void InsertImpl(std::weak\_ptr<Interval>) = 0;

virtual std::list<std::weak\_ptr<Interval>> FindAllIntersectionsImpl(std::weak\_ptr<Interval>) = 0;

protected:

uint size\_;

};

* **circular\_array:** aka [circular buffer](https://en.wikipedia.org/wiki/Circular_buffer), it offers the implementation of a circular array and its functions. For the moment it can only handle int and double types. Its usage is not recommended.

typedef struct \_circ\_array{

int length;

int count;

int start;

void \*data;

}circ\_array\_t;

* **hashtable:** it offers the implementation of a hashtable and its functions. Its usage is not recommended and you should prefer the use of std::unordered\_map of the C++ STL.

typedef struct \_hashtable\_elt\_t {

void \*key;

void \*value;

struct \_hashtable\_elt\_t \*previous;

struct \_hashtable\_elt\_t \*next;

} hashtable\_elt\_t;

typedef struct \_deprecated\_hashtable\_t {

int size;

hash\_function\_t hash;

equal\_function\_t equal;

clean\_function\_t clean;

clone\_function\_t clone;

hashtable\_elt\_t\* elements[HASHTABLE\_SIZE];

} hashtable\_t;

* **heap:** it offers the implementation of a heap and its functions. Its usage is not recommended and you should prefer the use of std::priority\_queue of the C++ STL.

typedef struct \_heap\_elt {

void \*key;

void \*data;

} heap\_elt\_t;

typedef struct \_heap {

int m\_capacity;

int capacity;

int size;

heap\_compare\_t compare;

heap\_elt\_t \*elts;

} heap\_t;

* **list:** it offers the implementation of a list and its functions. Its usage is not recommended and you should prefer the use of std::list of the C++ STL.

typedef struct \_list\_elt\_t {

void \*data;

struct \_list\_elt\_t \*next;

struct \_list\_elt\_t \*previous;

} list\_elt\_t;

typedef struct \_list\_t {

int size;

list\_elt\_t \*trav;

list\_elt\_t \*elements;

list\_elt\_t \*elements\_end;

} list\_t;

* **mem\_fs:** it offers the implementation of a fixed size memory management module and its functions. Its usage is not recommended and you should prefer the use of smart pointer of the C++ STL and/or malloc of C.
* **s**liding\_window : it offers the implementation of a sliding window and its functions. Its usage is not recommended as it had one specific case scenario in mind.

typedef struct \_sliding\_window{

int max\_size;

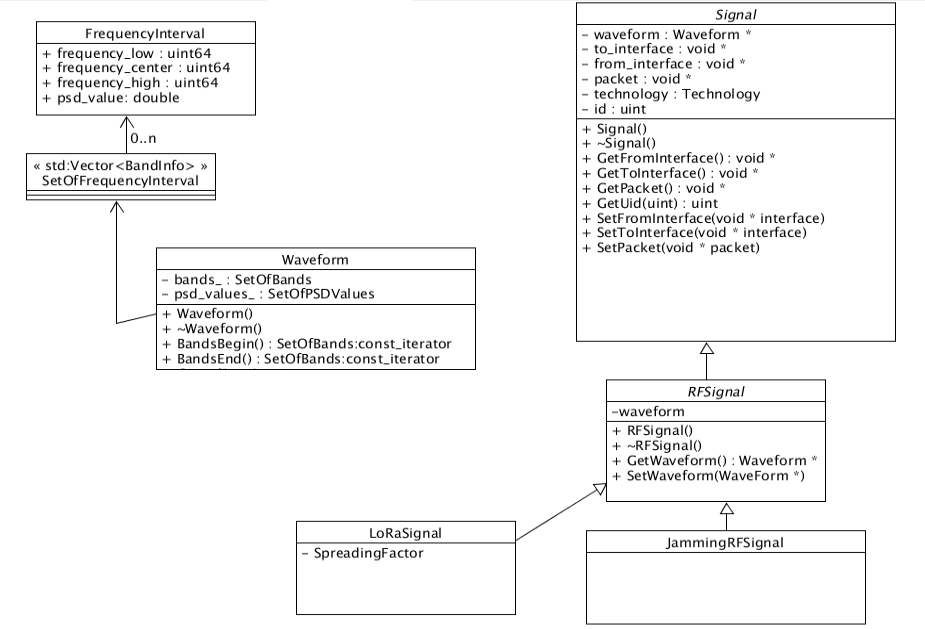
list\_t \*elts;

int data\_type;

} sliding\_window\_t;

### Types

This folder and the files inside it are used to define a set of fundamental types and classes that are used and exchanged throughout the simulations. The figure below gives a glimpse on the relationship of the different classes defined in types.



#### Packets

A radio packet follow this structure

typedef struct \_packet {

packetid\_t id; /\*\*< the packet id \*\*/

int size; /\*\*< size of the packet data \*\*/

int real\_size; /\*\*< real size of the packet data \*\*/

int type; /\*\*< type of the packet (for multistandard nodes) \*\*/

uint64\_t clock0; /\*\*< packet rx start \*\*/

uint64\_t clock1; /\*\*< packet rx end \*\*/

uint64\_t duration; /\*\*< packet tx/rx duration \*\*/

nodeid\_t node; /\*\*< node that has created the packet \*\*/

classid\_t interface; /\*\*< interface that has emitted the packet \*\*/

double txdBm; /\*\*< tx power in dBm \*\*/

int channel;

classid\_t modulation; /\*\*< modulation class \*\*/

uint64\_t Tb; /\*\*< radio bandwidth: time to send a bit \*\*/

destination\_t destination; /\*\*< destination for encapsulation in lower layer \*\*/

int frame\_type; /\*\*<\* type of access that is desired by layer (0 - S-CSMA, 1 - GTS, 2 - S-ALOHA)\*/

int frame\_control\_type; /\*\*<\* type of frame shared between NTW and MAC layers (0 Normal, 1 control without ACK) \*/

double PER; /\*\*< packet error rate \*\*/

double rxdBm; /\*\*< rx power in dBm \*\*/

double rxmW; /\*\*< rx power in mW \*\*/

double \*noise\_mW; /\*\*< packet noise in mW \*\*/

double \*ber; /\*\*< packet ber \*\*/

double RSSI; /\*\*< RSSI indicator in dBm \*\*/

double LQI; /\*\*<\* LQI indicator (value between 0(Bad Link quality) and 1(Good Link Quality)) \*/

double SINR; /\* Define the wideband SINR Value of the packet in linear \*/

hashtable\_t \*fields; /\*\*< packet data \*\*/

void \*signal; // pointer to the signal that contain the packet (temporary for backward compatibility)

} packet\_t;

#### Signals

The Signal is an abstract base class used to define a signal that will generated by a node and transmitted to others via the spectrum. It contains its beginning and end, the SNR and SINR values, along with source and destination nodes, and also the call\_t structures to and from of the interfaces, respectively transmitting and receiving.

/\*\* \brief The Abstract Base Class : Signal Class

\*

\* \fn GetBegin() return the time on which the signal begin

\* \fn GetEnd() return the time on which the signal ends

\* \fn SetDestination() set the nodeid\_t of the destination of the signal

\* \fn GetDestination() return the nodeid\_t of the destination of the signal

\* \fn GetSource() return the nodeid\_t of the source of the signal

\* \fn Clone() return a deep copy of the signal

\*\*/

class Signal {

public:

Signal (nodeid\_t source);

Signal (call\_t \*to,call\_t \*from\_interface);

Signal (call\_t \*to,call\_t \*from\_interface, Time T\_begin, Time T\_end);

virtual ~Signal();

void Print() const;

SignallUid GetUID() const;

std::shared\_ptr<Signal> Clone();

Time GetBegin() const;

Time GetEnd() const ;

double GetSINR() const;

double GetSNR() const;

void SetSINR(double SINR);

void SetSNR(double SNR);

call\_t GetTo\_Deprecated() const;

call\_t GetFrom\_Deprecated() const;

void SetTo\_Deprecated(call\_t to);

void SetFrom\_Deprecated(call\_t to);

void SetDestination(std::shared\_ptr<RegisteredRxNode> node\_id);

nodeid\_t GetSource() const;

void SetBegin(Time T\_begin);

void SetEnd(Time T\_end);

std::shared\_ptr<RegisteredRxNode> GetDestination() const;

void PrintSignal() const;

private:

virtual std::shared\_ptr<Signal> CloneImpl() = 0;

virtual void PrintSignalImpl() const = 0;

static SignallUid uid\_counter\_;

SignallUid uid\_;

protected:

call\_t to\_interface\_;

call\_t from\_interface\_;

Time begin\_;

Time end\_;

nodeid\_t source\_;

double SINR\_;

double SNR\_;

std::shared\_ptr<RegisteredRxNode> destination\_;

};

##### RFSignal

It describes a radio frequency signal with a waveform and a packet. It is a general RFSignal and can be used as it is or specialized as we will see below.

/\*\* \brief The Abstract Base Class : RFSignal Class

\*

\* \fn GetBegin() return the time on which the signal begin

\* \fn GetEnd() return the time on which the signal ends

\* \fn SetDestination() set the nodeid\_t of the destination of the signal

\* \fn GetDestination() return the nodeid\_t of the destination of the signal

\* \fn GetSource() return the nodeid\_t of the source of the signal

\* \fn Clone() return a deep copy of the signal

\*\*/

class RFSignal : public Signal{

public:

RFSignal(nodeid\_t source, packet\_t\* packet);

RFSignal (call\_t \*to,call\_t \*from\_interface, packet\_t\* packet);

RFSignal(nodeid\_t source, std::shared\_ptr<Waveform> waveform);

RFSignal (call\_t \*to,call\_t \*from\_interface, packet\_t\* packet, Time T\_begin, Time T\_end);

virtual ~RFSignal();

std::weak\_ptr<Waveform> GetWaveform() const;

Frequency GetBandwidth() const;

packet\_t \*GetPacket\_Deprecated();

void SetWaveform(std::shared\_ptr<Waveform> waveform);

private:

Frequency GetBandwidthImpl() const;

std::weak\_ptr<Waveform> GetWaveformImpl() const;

void SetWaveformImpl(std::shared\_ptr<Waveform>);

std::shared\_ptr<Signal> CloneImpl();

packet\_t \*GetPacket\_DeprecatedImpl();

void PrintSignalImpl() const;

protected:

std::shared\_ptr<Waveform> waveform\_;

packet\_t \*packet\_;

};

###### LoRaSignal

It is a specific RFSignal that describes a LoRa signal with its bandwidth, frequency center, spreading factor and all that is inherited.

/\*\* \brief The Concrete Derived Class : LoRa Class

\*

\* Besides the waveform, Lora Signals have the spreading factor of the signal

\*

\* \fn GetWaveformImpl() return the waveform of the lora signal

\* \fn SetWaveformImpl() sets the waveform of the lora signal

\* \fn CloneImpl() return a deep copy of the lora signal

\* \fn GetSpreadingFactor() return the spreading factor;

\* \fn SetSpreadingFactor() set the spreading factor;

\*\*/

class LoRaSignal : public RFSignal, public std::enable\_shared\_from\_this<LoRaSignal>{

public:

LoRaSignal(nodeid\_t source) : RFSignal(source, nullptr) {};

~LoRaSignal(){};

LoRaSignal(call\_t \*to, call\_t \*from\_interface, SpreadingFactor sf,Frequency freq\_center,

Frequency bandwidth, packet\_t \*packet) : RFSignal(to,from\_interface,packet) , sf\_(sf), freq\_center\_(freq\_center), bandwidth\_(bandwidth){};

LoRaSignal(nodeid\_t source, SpreadingFactor sf, Frequency freq\_center,

Frequency bandwidth, packet\_t \*packet) : RFSignal(source,packet) , sf\_(sf), freq\_center\_(freq\_center), bandwidth\_(bandwidth){};

void SetSpreadingFactor(SpreadingFactor sf){

sf\_=sf;

};

SpreadingFactor GetSpreadingFactor() const {return sf\_;};

Frequency GetFrequencyCenter() const {return freq\_center\_;} ;

private:

std::weak\_ptr<Waveform> GetWaveformImpl() const{ return waveform\_;};

std::shared\_ptr<Signal> CloneImpl(){

auto new\_signal = std::make\_shared<LoRaSignal>(source\_,sf\_, freq\_center\_, bandwidth\_,packet\_clone(packet\_));

new\_signal->SetWaveform(waveform\_->Clone());

waveform\_->SetSignal(new\_signal);

return new\_signal;

};

packet\_t \*GetPacket\_DeprecatedImpl() {return packet\_;};

void SetWaveformImpl(std::shared\_ptr<Waveform> waveform) {waveform\_ = waveform;};

void PrintSignalImpl() const {

std::cout<<"This is a LoRa Signal with SF = "<< sf\_ << std::endl;

waveform\_->Print();

};

Frequency GetBandwidthImpl() const{return bandwidth\_;}

SpreadingFactor sf\_ = 0;

Frequency freq\_center\_ = 0;

Frequency bandwidth\_ = 0;

};

###### JammingSignal

It is a specific RFSignal that describes a Jamming signal with all that is inherited. Jamming RF Signals do not need a packet as they only generate noisy waveforms.

/\*\* \brief The Concrete Derived Class : JammingRFSignal Class

\*

\* Jamming RF Signals do not need a packet as they only generate noisy waveforms

\*

\* \fn GetWaveformImpl() return the waveform of the jamming signal

\* \fn SetWaveformImpl() sets the waveform of the jamming signal

\* \fn CloneImpl() return a deep copy of the jamming signal

\*\*/

class JammingRFSignal : public RFSignal{

public:

JammingRFSignal(nodeid\_t source) : RFSignal(source, nullptr){};

~JammingRFSignal(){};

private:

std::weak\_ptr<Waveform> GetWaveformImpl(){ return waveform\_;};

void PrintSignalImpl() const {

std::cout<<"This is a Jamming Signal "<<std::endl;

waveform\_->Print();

};

std::shared\_ptr<Signal> CloneImpl(){

auto new\_signal = std::make\_shared<JammingRFSignal>(source\_);

new\_signal->SetWaveform(waveform\_->Clone());

waveform\_->SetSignal(new\_signal);

return new\_signal;

};

packet\_t \*GetPacket\_DeprecatedImpl() {return nullptr;};

void SetWaveformImpl(std::shared\_ptr<Waveform> waveform) {waveform\_ = waveform;};

};

#### Interval

The interval is an abstract class used to define an interval with low and high boundaries, which define the beginning and end of the interval.

/\*\* \brief The Abstract Base Class : Interval Class

\* This means that no instance of the Interval class can exist.

\* Only classes which inherit from the Interval class can exist.

\*

\* \fn GetLowPoint return the lowest point of the interval

\* \fn GetHighPoint return the highest point of the interval

\* \fn GetUID return the UID of the interval

\*\*/

class Interval {

public:

Interval();

virtual ~Interval() {};

IntervalBoundary GetLowPoint() const;

IntervalBoundary GetHighPoint() const;

void Print() const;

IntervalUid GetUID();

private:

virtual IntervalBoundary GetLowPointImpl() const = 0;

virtual IntervalBoundary GetHighPointImpl() const = 0;

virtual void PrintImpl() const;

static IntervalUid uid\_counter\_;

IntervalUid uid\_;

};

##### FrequencyInterval

It describes a frequency interval, with a low, high boundaries and center point. It is used whenever you need to define a frequency interval.

/\*\* \brief The Concrete Base Class : FrequencyInterval Class

\*

\* \fn GetLowPoint return the low value of the frequency interval

\* \fn GetHighPoint return the high value of the frequency interval

\* \fn GetCenter return the frequency center

\* \fn Clone return a cloned frequency interval

\*\*/

class FrequencyInterval : public Interval {

public:

FrequencyInterval(const Frequency low,const Frequency high);

FrequencyInterval(const Frequency low,const Frequency high,const Frequency center);

Frequency GetCenter() const;

std::shared\_ptr<FrequencyInterval> Clone();

private:

virtual std::shared\_ptr<FrequencyInterval> CloneImpl();

void PrintImpl() const;

protected:

Frequency GetLowPointImpl() const;

Frequency GetHighPointImpl() const;

Frequency low\_;

Frequency high\_;

Frequency center\_;

};

##### FrequencyIntervalWaveform

It describes a frequency interval used in a waveform, with a low, high boundaries, center point(inherited), the PSD value of this specific frequency interval and a link to the waveform it belongs to. It is used whenever you need to define a waveform with its several frequency intervals.

/\*\* \brief The Concrete Derived Class : FrequencyIntervalWaveform Class

\* It is a specialization of the FrequencyInterval Class that

\* is used to create an waveform. In fact, one of its member function

\* is a weak\_ptr to the waveform it belongs.

\*

\* \fn GetLowPoint return the low value of the frequency interval

\* \fn GetHighPoint return the high value of the frequency interval

\* \fn GetCenter return the frequency center

\* \fn GetPSDValue return the PSD value of the frequency interval

\* \fn SetPSDValue changes the PSD value of the frequency interval

\* \fn CloneImpl return a copy of the frequency interval

\*\*/

class FrequencyIntervalWaveform : public FrequencyInterval {

public:

FrequencyIntervalWaveform(const Frequency low,const Frequency high,const Frequency center,

const PSDValue psd\_value, std::weak\_ptr<Waveform> waveform );

FrequencyIntervalWaveform(const Frequency low,const Frequency high,const Frequency center,

const PSDValue psd\_value );

void AddToWaveform(std::weak\_ptr<Waveform> waveform);

std::weak\_ptr<Waveform> GetWaveform();

PSDValue GetPSDValue() const;

void SetPSDValue(PSDValue psd\_value);

private:

std::shared\_ptr<FrequencyInterval> CloneImpl();

void PrintImpl() const;

std::weak\_ptr<Waveform> waveform\_ptr\_;

PSDValue psd\_value\_;

};

##### FrequencyIntervalRegisteredRxNode

It describes a frequency interval used by a registered node, with a low, high boundaries, center point (inherited), and a link to the RegisteredRxNode it belongs to. It is used whenever you need to define a RegisteredRxNode with its several frequency intervals.

class FrequencyIntervalRegisteredRxNode : public FrequencyInterval {

public:

FrequencyIntervalRegisteredRxNode(const Frequency low,const Frequency high,const Frequency center,std::weak\_ptr<RegisteredRxNode> rx\_node\_filter );

FrequencyIntervalRegisteredRxNode(const Frequency low,const Frequency high, const Frequency center);

void AddToRxNode(std::weak\_ptr<RegisteredRxNode> RxNode);

std::weak\_ptr<RegisteredRxNode> GetRxNode() const;

private:

std::shared\_ptr<FrequencyInterval> CloneImpl();

void PrintImpl() const;

std::weak\_ptr<RegisteredRxNode> registered\_rx\_node\_ptr\_;

};

##### TimeInterval

It describes a time interval, with a low and a high boundaries. It is used whenever you need to define a time interval.

/\*\* \brief The Concrete Base Class : TimeInterval Class

\*

\* \fn GetLowPoint return the low value of the time interval

\* \fn GetHighPoint return the high value of the time interval

\* \fn Clone return a cloned time interval

\*\*/

class TimeInterval : public Interval {

public:

TimeInterval(const Time low,const Time high);

std::shared\_ptr<TimeInterval> Clone();

private:

virtual std::shared\_ptr<TimeInterval> CloneImpl();

void PrintImpl() const;

protected:

Time GetLowPointImpl() const;

Time GetHighPointImpl() const;

Time low\_;

Time high\_;

};

##### RegisteredRxNode

It describes a node registered on the spectrum, with a set of frequencies it is listening as well as the PHY model it is connected.

class RegisteredRxNode{

public:

RegisteredRxNode(SetOfFrequencyIntervalRegisteredRxNode freqs,nodeid\_t node\_id, PhyModel\* phy\_model);

RegisteredRxNode(SetOfFrequencyIntervalRegisteredRxNode freqs ,nodeid\_t node\_id);

~RegisteredRxNode();

nodeid\_t GetNodeID() const;

void AddFrequencyInterval(std::shared\_ptr<FrequencyIntervalRegisteredRxNode> freq);

PhyModel\* GetPhyModel() const;

SetOfFrequencyIntervalRegisteredRxNode GetAllFrequencyInterval() const;

private:

SetOfFrequencyIntervalRegisteredRxNode frequency\_bands\_;

nodeid\_t node\_id\_;

PhyModel\* phy\_model\_ptr\_;

};

#### Waveform

It basically describes a waveform with a set of frequency intervals that gives shape to the waveform along with a weak pointer to the signal it belongs.

/\*\* \brief The Base Class : Waveform Class

\* Although it is not necessary, this class can be inherited to

\* create another specific waveform class.

\*

\* \fn AddFrequencyInterval - inserts a frequency interval in the waveform

\* \fn GetAllFrequencyInterval - return all the frequencies intervals that describe the waveform

\* \fn GetUID - return the UID of the waveform

\* \fn Clone return a cloned waveform

\*\*/

class Waveform : public std::enable\_shared\_from\_this<Waveform>{

public:

Waveform(SetOfFrequencyIntervalWaveform frequency\_bands);

Waveform();

~Waveform();

void AddFrequencyInterval(std::shared\_ptr<FrequencyIntervalWaveform> freq);

SetOfFrequencyIntervalWaveform GetAllFrequencyInterval();

std::shared\_ptr<Waveform> Clone();

WaveformUid GetUID() const;

std::weak\_ptr<Signal> GetSignal();

void SetSignal(std::weak\_ptr<Signal>);

void Print() const;

private:

WaveformUid uid\_; // maybe we do not need uid counter, we can check the memory of each object and this will be the UID;

static WaveformUid uid\_counter\_;

SetOfFrequencyIntervalWaveform frequency\_interval\_; // the set of frequency intervals that describes the waveform

std::weak\_ptr<Signal> signal\_; // pointer to the signal of which it belongs

};

### Functions

#### Simulation

Several functions have been defined for simulation purpose:

* **get\_time**: It’s a function to return the current simulation time. Defined in kernel/definitions/scheduler.c&h files

uint64\_t get\_time ( void );

* **scheduler\_get\_end**: This function returns the date of the end of the simulation. Defined in kernel/definitions/scheduler.c&h files

uint64\_t scheduler\_get\_end(void);

* **get\_node\_count**: This function returns the number of simulated nodes. Defined in kernel/definitions/node.c&h files

int get\_node\_count ( void );

* **get\_topology\_area**: This function returns the size of the network area. Defined in kernel/model\_handlers/topology.c&h files

position\_t \* get\_topology\_area ( void );

#### Class

Several functions have been defined in kernel/definitions/class.c&h files:

* **get\_class\_private\_data**: This function gets the private (global) memory of the class and returns its void pointer. This memory is shared by all nodes using the same class.

void \*get\_class\_private\_data(call\_t \*to);

* **set\_class\_private\_data**: This function sets the private (global) memory of the class.

void set\_class\_private\_data(call\_t \*to, void \*data);

* **get\_class\_bindings\_up**: This function returns the classes that are up “to->class” in “to->object”

array\_t \*get\_class\_bindings\_up(call\_t \*to);

* **get\_class\_bindings\_down**: This function returns the classes that are down “to->class” in “to->object”

array\_t \*get\_class\_bindings\_down(call\_t \*to);

For example, from call\_t “to” composed of {class\_id, object\_id} with a classid MAC, we can access to its transceiver down classid using get\_class\_bindings\_down.

array\_t \*down = get\_class\_bindings\_down(to);

call\_t to0 = {down->elts[i], to->object};

If there is several down class, the first one will be down->elts[0], the second one down->elts[1]… The order is the same that in the XML configuration file.

#### Node

Several functions have been defined in kernel/definitions/node.c&h files:

* **get\_node\_position**: This function uses the node id as parameter and returns the node position. This function is used for read only.

position\_t \*get\_node\_position(nodeid\_t node);

* **get\_node\_by\_id**: This function is used to return the address of a node from its id.

node\_t \*get\_node\_by\_id(nodeid\_t id);

* **get\_node\_private\_data**: This function gets the private (local) memory of the node and returns its void pointer.

void \*get\_node\_private\_data(call\_t \*to);

* **set\_node\_private\_data**: This function sets the private (local) memory of the node.

void set\_node\_private\_data(call\_t \*to, void \*data);

* **node\_kill**: This function kills a node during the simulation.

void node\_kill(nodeid\_t id);

* **is\_node\_alive**: This function is used to check whether a node is alive. 1 if the node is alive and 0 otherwise.

int is\_node\_alive(nodeid\_t id);

#### Shared Data

A shared data mechanism has been implemented for a cross-layer approach. As modules are implemented separately and do not share information, this shared memory makes available some information for all the layers. An example how to access to this memory is provides hereafter.

nodedata->shared\_data = (shared\_data\_t \*) (get\_class\_by\_name("shared"))->methods->shared.get\_shared\_data(to->object);

#### Packet

Several functions have been defined in kernel/definitions/packet.c&h files in order to manipulate packet structure:

* **packet\_create**: This function allocates a packet with real size argument and returns the newly allocated packet

packet\_t \*packet\_create(call\_t \*to, int size, int real\_size);

* **packet\_dealloc:** This function deallocates a packet

void packet\_dealloc(packet\_t \*packet);

* **packet\_clone**: This function duplicates a packet. Data and other information are copied. This function takes in parameter the packet to clone and returns the cloned packet.

packet\_t \*packet\_clone(packet\_t \*packet);

* **packet\_add\_field**: This function adds a new field into a packet and attributes a name to this field.

void packet\_add\_field(packet\_t \*packet, char \*name, field\_t \*field);

* **packet\_retrieve\_field**: This function extracts a field from a packet according to its name.

field\_t \*packet\_retrieve\_field(packet\_t \*packet, char \*name);

* **packet\_retrieve\_field\_value\_ptr**: This function returns the pointer of a field contained in a packet according to its name.

void \*packet\_retrieve\_field\_value\_ptr(packet\_t \*packet, char\* name);

Hereafter we provide an example how to create a packet in the application layer:

* Create a structure for the application header

struct data\_packet\_header {

int source; /\*!< Specificies the ID of the source of the generated data packet. \*/

int sequence; /\*!< Specificies the sequence number of the generated data packet. \*/

double field\_n; /\*!< any kind of field can be defined \*/

};

* Create a packet

packet\_t \*packet = packet\_create(to, 0, 0);

* Allocate the memory of the header and create a new field

int data\_packet\_header\_size = sizeof(struct data\_packet\_header);

struct data\_packet\_header \*header = malloc(data\_packet\_header\_size);

field\_t \*field\_data\_packet\_header = field\_create(INT, data\_packet\_header\_size, header);

* Fill the header with the appropriate information

header->source = to->object;

header->sequence = nodedata->sequence++;

…

* Update the size of the packet

data\_packet\_header\_size = 9;

packet->size += data\_packet\_header\_size;

packet->real\_size += data\_packet\_header\_size\*8;

* insert the new field containing the header into the packet and provide it a FIELD\_NAME

packet\_add\_field(packet, "FIELD\_NAME", field\_data\_packet\_header);

At the reception, the field of the packet can be extracted:

struct data\_packet\_header \*header = (struct data\_packet\_header \*) packet\_retrieve\_field\_value\_ptr(packet, " FIELD\_NAME ");

or using :

field\_t \*field\_header = packet\_retrieve\_field(packet, "FIELD\_NAME ");

struct data\_packet\_header \*header;

header = (struct data\_packet\_header \*) field\_getValue(field\_header);

#### Signal

In this section, we describe an example how to create a signal and how to use it.

You can generate and a simple frequency interval:

auto freq\_min = freq\_center - bandwidth/2;

auto freq\_max = freq\_center + bandwidth/2;

auto psd = dBm2mW(power\_dBm) / bandwidth;

auto frequency\_intervals\_tx\_ = std::make\_shared<FrequencyIntervalWaveform>(freq\_min, freq\_max, freq\_center, psd);

Create the waveform and the new signal that contains that frequency interval:

auto waveform = WaveformFactory::CreateWaveform(freq\_intervals);

auto new\_signal = RFSignalFactory::CreateSignal(to, from\_interface,waveform,packet);

You can also update the signal whenever needed (usually after applying pathloss, etc):

for (auto freq\_interval : rf\_signal->GetWaveform().lock()->GetAllFrequencyInterval()){

auto psd = rf\_signal->GetPacket\_Deprecated()->rxmW / (freq\_interval->GetHighPoint() - freq\_interval->GetLowPoint());

freq\_interval->SetPSDValue(psd);

At the reception, we receive a signal from which we can get some informations

signal->GetUID()

signal->GetEnd()

signal->GetBegin()

At the reception of a signal, we can start to check if the received object has the expected type

auto rf\_signal = std::dynamic\_pointer\_cast<RFSignal>(signal);

if (rf\_signal){

We can get some informations from the rf\_signal

rf\_signal->GetBandwidth()

Recuperate the packet of the signal

packet\_t \*packet = rf\_signal->GetPacket\_Deprecated();

or update the packet information

rf\_signal->GetPacket\_Deprecated()->rxmW;

Recuperate the to and from of the signal

call\_t to = {rf\_signal->GetTo\_Deprecated().classid, rf\_signal->GetTo\_Deprecated().object};

call\_t from = {rf\_signal->GetFrom\_Deprecated().classid, rf\_signal->GetFrom\_Deprecated().object};

Update the information of the signal

rf\_signal->SetSNR(packet->rxmW/noise\_mW);

rf\_signal->SetSINR(packet->rxmW/noise\_mW);

#### Scheduler 3.1

The scheduler is event-oriented. An event is used to call a callback function of to manage a packet with the following structure:

typedef struct \_event {

uint64\_t clock; /\* event time \*/

int priority; /\* event priority \*/

int id; /\* event id \*/

union {

struct {

call\_t to;

call\_t from;

callback\_t callback;

void \*arg;

} cb;

struct {

packet\_t \*packet;

call\_t to;

call\_t from;

} rx;

nodeid\_t nodeid;

} u;

} event\_t;

The RX begin, RX end and TX end events are directly bring to the scheduler by media\_rxtx kernel function.

void scheduler\_add\_rx\_begin(uint64\_t clock, call\_t \*to, call\_t \*from, packet\_t \*packet);

void scheduler\_add\_rx\_end(uint64\_t clock, call\_t \*to, call\_t \*from, packet\_t \*packet);

void scheduler\_add\_tx\_end(uint64\_t clock, call\_t \*to, call\_t \*from, packet\_t \*packet);

Each mobility model must manage its update event using scheduler\_add\_mobility function. This function adds a mobility event to the scheduler with a mobility date. When this event happens, the kernel calls update\_position method of the mobility model.

void scheduler\_add\_mobility(uint64\_t clock);

The function scheduler\_add\_callback schedules the callback of a function at a given time. The parameters are the time of the callback, the calls to and from, the callback function and the parameter of this callback.

event\_t \*scheduler\_add\_callback(uint64\_t clock, call\_t \*to, call\_t \*from, callback\_t callback, void \*arg);

During a simulation, it could be necessary to unschedule some scheduled events. Then the function scheduler\_delete\_callback can delete an event from the scheduler.

void scheduler\_delete\_callback(event\_t \*event);

As an example, we can schedule a new callback function with the following code.

scheduler\_add\_callback(get\_time()+Duration, to, from, callback\_function, NULL);

#### Scheduler 4.0

The new version of the scheduler has a base class Scheduler that needs to be implemented. The kernel call the public methods of the Scheduler class.

/\*\* \brief Scheduler: The Scheduler abstract base class

\*

\* \fn AddBirth() add a birth event

\* \fn AddQuit() add a quit event (end of simulation)

\* \fn AddMobility() add a mobility event

\* \fn AddMilestone() add a milestone event

\* \fn AddTxEnd() add a tx\_end event

\* \fn AddRxBegin() add a rx\_begin event

\* \fn AddRxEnd() add a rx\_end event

\* \fn AddCallback() add a rx\_end event

\* \fn AddRxEnd() add a callback event

\* \fn DeleteCallback() delete a callback event

\* \fn NextEvent() return the next event to be executed

\* \fn CountEvents() return the number of events to be executed

\* \fn CountEventsExecuted() return the number of events already executed

\* \fn AddEventImpl() add an event - to be implemented

\* \fn DeleteEventImpl() delete an event - to be implemented

\* \fn NextEventImpl() return the next event to be executed - to be implemented

\* \fn CountEventsImpl() return the number of events to be executed - to be implemented

\*\*/

class Scheduler{

**public**:

Scheduler();

Scheduler(Time end);

virtual ~Scheduler(){};

void SimulationRun();

void SimulationEnd();

Time SimulationTimeGet();

Time SimulationTimeGetEnd();

void SimulationTimeSetEnd(Time end);

void SimulationTimeAdvanceClock(Time clock);

event\_t AddBirth(Time clock, nodeid\_t id);

event\_t AddQuit(Time clock);

event\_t AddMobility(Time clock);

event\_t AddMilestone(Time clock);

event\_t AddTxEnd(Time clock, call\_t to, call\_t from, packet\_t \*packet);

event\_t AddRxBegin(Time clock, call\_t to, call\_t from, packet\_t \*packet);

event\_t AddRxEnd(Time clock, call\_t to, call\_t from, packet\_t \*packet);

event\_t AddTxEnd(Time clock, SpectrumModel \*spectrum, std::shared\_ptr<Signal> signal);

event\_t AddRxBegin(Time clock, SpectrumModel \*spectrum, std::shared\_ptr<Signal> signal);

event\_t AddRxEnd(Time clock, SpectrumModel \*spectrum, std::shared\_ptr<Signal> signal);

event\_t AddCallback(Time clock, call\_t to, call\_t from, callback\_t callback, void \*arg);

void DeleteCallback(event\_t event\_info);

int CountEvents();

int CountEventsExecuted();

**protected**:

bool running\_;

Time clock\_;

Time end\_;

int nbr\_events\_executed\_;

**private**:

void ExecuteEvent(std::unique\_ptr<Event> e);

event\_t AddEvent(std::unique\_ptr<Event> e);

void DeleteEvent(uid\_t uid);

std::unique\_ptr<Event> NextEvent();

virtual int CountEventsImpl() = 0;

virtual void AddEventImpl(std::unique\_ptr<Event> e) = 0;

virtual void DeleteEventImpl(uid\_t uid) = 0;

virtual std::unique\_ptr<Event> NextEventImpl(void) = 0;

};

For backward compatibility, we kept the interfaces of the version 3.1 described above. This means that whenever a call for scheduler\_add\_callback is done, we will actually redirect it to the AddCallback method of the Scheduler class.

##### Events

The scheduler is still event-oriented. An event is used to call a callback function of to manage a packet with the following structure:

typedef struct \_event {

uid\_t uid\_; // event id

uint64\_t clock\_; // event time

} event\_t;

Un C++ Implementation, the structure event is the following:

struct Event{

Time clock\_; // event time

event\_priority\_t priority\_; // event priority

EventUid uid\_; // event id

static EventUid uid\_counter\_;

Event();

Event(Time clock, event\_priority\_t priority) ;

bool operator >(const Event &rhs);

bool operator <(const Event &rhs) ;

bool operator ==(const Event &rhs) ;

};

4 events are derived:

* EventBirth is the event for node birth
* EventCallback is the event managing the callback
* EventRxTx is the event for RX\_begin, RX\_end, TX\_begin and TX\_end.
* EventRxTxSignal is the event for RX\_signal\_begin, RX\_ signal\_end, TX\_ signal\_begin and TX\_ signal\_end.

struct EventBirth : public Event{

nodeid\_t nodeid\_;

EventBirth(Time clock, event\_priority\_t priority, nodeid\_t nodeid);

};

struct EventCallback : public Event{

call\_t to\_;

call\_t from\_;

callback\_t callback\_;

void \*arg\_;

EventCallback(Time clock, event\_priority\_t priority, call\_t to, call\_t from, callback\_t callback, void \*arg);

};

struct EventRxTx : public Event{

call\_t to\_;

call\_t from\_;

packet\_t \*packet\_;

EventRxTx(Time clock, event\_priority\_t priority, call\_t to, call\_t from, packet\_t \*packet);

};

struct EventRxTxSignal : public Event{

std::shared\_ptr<Signal> signal\_;

SpectrumModel \*spectrum\_;

EventRxTxSignal(Time clock, event\_priority\_t priority, SpectrumModel \*spectrum, std::shared\_ptr<Signal> signal) ;

};

Note that it is possible to link C and C++ events thanks to UID

##### Scheduler using STD containers

Furthermore, we provided an implementation of the scheduler using C++ standard containers. In such implementation we use use a priority queue for all events along with an unordered set used to store the uid of deleted\_callbacks. Thus, whenever a callback event is the next, we check whether it has been deleted or not by looking for its uid inside the unordered set. (O(1) complexity - considering our hash won't have collisions).

Notice that we could also use only a priority queue, but this would imply that while deleting a callback we would need to either:

* pop elements until you find and then insert them again
* inherit the std::priority\_queue and find the element on the protected container, delete it, and make a heap with the remaining elements.

Both solutions would have a higher time complexity (O(nlogn) at least). The extra memory needed is low compared to time gains, once we are only storing the uid's of the events that have been deleted and after their time expiration, we clear them from the unordered set.

The definition of the class can be found below.

/\*\* \brief SchedulerStandardContainers: The Scheduler implementation using C++'s STD containers

\*

\* \fn AddEventImpl() add an event in the priority queue

\* \fn DeleteEventImpl() delete an event (mark an event as deleted on the unordered set)

\* \fn NextEventImpl() return the next event to be executed

\* \fn CountEventsImpl() return the number of events on the queue

\*\*/

class SchedulerStandardContainers : public Scheduler {

**public:**

SchedulerStandardContainers();

SchedulerStandardContainers(Time end);

~SchedulerStandardContainers();

**private:**

void AddEventImpl(std::unique\_ptr<Event> e\_);

void DeleteEventImpl(uid\_t uid);

std::unique\_ptr<Event> NextEventImpl();

int CountEventsImpl();

std::priority\_queue<std::unique\_ptr<Event>, std::vector<std::unique\_ptr<Event>>, CompareEventGreater> events\_queue\_;

std::unordered\_set<uid\_t> deleted\_events\_;

};

#### Timer

In librairies/timer, some functions are implemented to model timer behaviors.

The timer structure is the following :

typedef struct qtimer\_s {

void \*trigger\_parameters; //parameters for next\_trigger functions (eg: period for periodic timer)

int (\*conditional\_end)(call\_t \*to, void \*timer\_id); //pointer to a function that returns 1 if the timer must be destroyed and 0 otherwise

void (\*callback\_function)(call\_t \*to, void \*timer\_id); // function to callback when the timer triggers.

uint64\_t (\*next\_trigger)(call\_t \*to, void \*timer\_id);

call\_t \*to;

}qtimer\_t;

The common timer next trigger functions are:

* a constant value given as parameter.

uint64\_t periodic\_trigger(call\_t \*to, void \*timer\_id);

* an exponential sequence of form initial\_value\*ration^rank - offset

uint64\_t exponential\_trigger(call\_t \*to, void \*timer\_id);

* a random time chosen in the given interval.

uint64\_t uniform\_random\_trigger(call\_t \*to, void \*timer\_id);

The common conditional end function is:

int never\_stop(call\_t \*to, void \*timer\_id);

The main functions are:

* create a new timer

void \*create\_timer(call\_t \*to, void \*callback\_function, void \*conditional\_end, void \*next\_trigger, void \*trigger\_parameters);

* start a newly created timer

void \*start\_timer(void \*timer\_id, uint64\_t delay);

* destroy a timer

void destroy\_timer(void \*timer\_id);

* change the timer parameter value

void change\_parameter(void \*timer\_id, void \*new\_parameters);

* fetch a timer

qtimer\_t \*fetch\_timer(void \*timer\_id);

As an example, we can manage the timer with the following code.

void \*timer\_id;

uint64\_t start = get\_time();

we create a periodic timer for the function callback\_function

timer\_id = create\_timer(to, callback\_function, never\_stop, periodic\_trigger, period);

we start the timer

start\_timer(timer\_id, start);

At the end, we destroy the timer

destroy\_timer(nodedata->timer\_id);

#### Model handler functions

In kernel/model\_handlers, methods of several models are linked.

##### Energy

In kernel/model\_handlers/energy, some declarations are used to link energy methods:

typedef struct \_energy\_methods {

void (\*recharge) (call\_t \*to, double energy);

void (\*consume) (call\_t \*to, double current, uint64\_t duration);

double (\*energy\_consumed) (call\_t \*to);

double (\*energy\_recharged) (call\_t \*to);

double (\*energy\_remaining) (call\_t \*to);

double (\*energy\_status) (call\_t \*to);

double (\*get\_supply\_voltage) (call\_t \*to);

} energy\_methods\_t;

These declarations have been defined to ease the usage of methods whatever the energy model then to keep the modularity of the implementation.

The declaration usable by any models are the following:

* **energy\_recharge** is called to calculate the amount of energy to be recharged. The parameter is the amount of energy to be recharged (in J).

void energy\_recharge(call\_t \*to, double energy\_J);

* **energy\_consume** is the generic function called to add the energy consumption associated to an operation. The parameters are the current used for the operation and the duration of the operation in ns.

void energy\_consume(call\_t \*to, double current\_mA, uint64\_t duration\_ns);

* **energy\_check\_energy\_consumed** returns the total amount of consumed energy.

double energy\_check\_energy\_consumed(call\_t \*to);

* **energy\_check\_energy\_recharged** returns the total amount of recharged energy.

double energy\_check\_energy\_recharged(call\_t \*to);

* **energy\_check\_energy\_remaining** is used to verify how much energy is still remaining. It returns the remaining energy in Joule.

double energy\_check\_energy\_remaining(call\_t \*to);

* **energy\_status** is used to verify the status of the battery in percentage. It returns the percentage of energy charge.

double energy\_check\_energy\_status(call\_t \*to);

* **energy\_get\_supply\_voltage** is used to get the supplied voltage. It returns the voltage in Volt.

double energy\_get\_supply\_voltage(call\_t \*to);

##### Interface

In kernel/model\_handlers/interface, some declarations are used to link interface/antenna methods:

typedef struct \_interface\_methods {

void (\*rx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

void (\*cs) (call\_t \*to, call\_t \*from, packet\_t \*packet);

double (\*get\_loss) (call\_t \*to, call\_t \*from);

angle\_t \* (\*get\_angle) (call\_t \*to, call\_t \*from);

void (\*set\_angle) (call\_t \*to, call\_t \*from, angle\_t \*angle);

double (\*gain\_tx) (call\_t \*to, call\_t \*from, position\_t \*pos);

double (\*gain\_rx) (call\_t \*to, call\_t \*from, position\_t \*pos);

mediumid\_t (\*get\_medium)(call\_t \*to, call\_t \*from);

int (\*get\_type) (call\_t \*to);

} interface\_methods\_t;

These declarations have been defined to ease the usage of methods whatever the antenna model used in order to keep the modularity of the implementation.

The declaration usable by any models are the following:

* **interface\_get\_loss** returns the signal loss induced by an interface circuit in dB

double interface\_get\_loss(call\_t \*to, call\_t \*from);

* **interface\_get\_angle** returns the interface orientation

angle\_t \*interface\_get\_angle(call\_t \*to, call\_t \*from);

* **interface\_set\_angle** sets the interface orientation

void interface\_set\_angle(call\_t \*to, call\_t \*from, angle\_t \*angle);

* **interface\_rx** forwards a received packet to the interface

void interface\_rx(call\_t \*to, call\_t \*from, packet\_t \*packet);

* **interface\_cs** launch the detection of a packet (carrier sensing and capture effect)

void interface\_cs(call\_t \*to, call\_t \*from, packet\_t \*packet);

* **interface\_gain\_tx** returns the tx interface gain towards the destination direction

double interface\_gain\_tx(call\_t \*to, call\_t \*from, position\_t \*position);

* **interface\_gain\_rx** returns the rx interface gain towards the destination direction

double interface\_gain\_rx(call\_t \*to, call\_t \*from, position\_t \*position);

* **interface\_get\_medium** returns the medium matching the node interface. This function is often used to know on which medium the antenna is connected.

mediumid\_t interface\_get\_medium(call\_t \*to, call\_t \*from);

These functions are mainly used in media\_rxtx file calculating the different loss of the antenna on the budget link.

##### Link

In kernel/model\_handlers/link, some declarations are used to link methods:

typedef struct \_link\_methods {

link\_condition\_t (\*get\_link\_condition) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

link\_type\_t (\*get\_link\_type) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

link\_communication\_type\_t (\*get\_communication\_type) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

double (\*get\_mutual\_orientation) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

int (\*get\_complementary\_link\_condition) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

} link\_methods\_t;

These declarations have been defined to ease the usage of methods whatever the link model used in order to keep the modularity of the implementation.

The declaration usable by any models are the following:

* **link\_get\_link\_condition** returns the condition of the link (whether is in LOS, NLOS, etc)

link\_condition\_t link\_get\_link\_condition(call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

* **link\_get\_communication\_type** returns the communication type that this link is describing (whether is in WiFi, 802.15.4, etc)

link\_communication\_type\_t link\_get\_communication\_type(call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

* **link\_get\_mutual\_orientation** returns the mutual orientation of two nodes, according to their position and antenna positionnig

double link\_get\_mutual\_orientation(call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

* **link\_get\_complementary\_link\_condition** returns the any complementary condition or information the link may have.

int link\_get\_complementary\_link\_condition(call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

These functions are mainly used by pathloss, fading and shadowing for calculating the different propagation loss according to the specificity of the link.

##### Modulation

In kernel/model\_handlers/modulations, some declarations are used to link modulation methods:

typedef struct \_modulation\_methods {

double (\* modulate) (call\_t \*to, call\_t \*from, double snr);

int (\* bit\_per\_symbol) (call\_t \*to, call\_t \*from);

int (\*get\_modulation\_type) (call\_t \*to, call\_t \*from);

void (\*set\_modulation\_type) (call\_t \*to, call\_t \*from, int modulation\_type);

} modulation\_methods\_t;

The declaration usable by any models are the following:

* **Modulation\_bit\_per\_symbol** returns the number of bit per symbol for a given modulation scheme

int modulation\_bit\_per\_symbol(classid\_t modulation);

* **do\_modulate** returns the bit error rate for a given modulation scheme. The parameters are the modulation, the received power and the noise power.

double do\_modulate(classid\_t modulation, double rxmW, double noise);

##### Transceiver

In kernel/model\_handlers/transceiver, some declarations are used to link transceiver methods:

typedef struct \_transceiver\_methods {

void (\*rx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

void (\*tx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

int (\*set\_header) (call\_t \*to, call\_t \*from, packet\_t \*packet, destination\_t \*dst);

int (\*get\_header\_size) (call\_t \*to, call\_t \*from);

int (\*get\_header\_real\_size) (call\_t \*to, call\_t \*from);

void (\*tx\_end) (call\_t \*to, call\_t \*from, packet\_t \*packet);

void (\*cs) (call\_t \*to, call\_t \*from, packet\_t \*packet);

double (\*get\_noise) (call\_t \*to, call\_t \*from);

double (\*get\_cs) (call\_t \*to, call\_t \*from);

double (\*get\_power) (call\_t \*to, call\_t \*from);

void (\*set\_power) (call\_t \*to, call\_t \*from, double power);

int (\*get\_channel) (call\_t \*to, call\_t \*from);

void (\*set\_channel) (call\_t \*to, call\_t \*from, int channel);

classid\_t (\*get\_modulation) (call\_t \*to, call\_t \*from);

void (\*set\_modulation) (call\_t \*to, call\_t \*from, classid\_t modulation);

uint64\_t (\*get\_Tb) (call\_t \*to, call\_t \*from);

uint64\_t (\*get\_Ts) (call\_t \*to, call\_t \*from);

void (\*set\_Ts) (call\_t \*to, call\_t \*from, uint64\_t Ts);

double (\*get\_sensibility) (call\_t \*to, call\_t \*from);

void (\*set\_sensibility) (call\_t \*to, call\_t \*from, double sensibility);

void (\*sleep) (call\_t \*to, call\_t \*from);

void (\*wakeup) (call\_t \*to, call\_t \*from);

void (\*switch\_rx) (call\_t \*to, call\_t \*from);

void (\*switch\_idle) (call\_t \*to, call\_t \*from);

int (\*get\_modulation\_bit\_per\_symbol) (call\_t \*to, call\_t \*from);

int (\*get\_modulation\_type) (call\_t \*to, call\_t \*from);

void (\*set\_modulation\_type) (call\_t \*to, call\_t \*from, int modulation\_type);

} transceiver\_methods\_t;

The declaration usable by any models are the following:

* **transceiver\_get\_cs** is used to perform the carrier sense mechanism. It returns the signal strength of the currently received signal or MIN\_DBM if no current signal

double transceiver\_get\_cs(call\_t \*to, call\_t \*from);

* **transceiver\_get\_noise** is used to perform the clear channel assessment mechanism. It returns the noise strength on the transceiver interface or MIN\_DBM if no current signal

double transceiver\_get\_noise(call\_t \*to, call\_t \*from);

* **transceiver\_get\_power** is used to get the transceiver tx power. It returns the current transmission power in dBm.

double transceiver\_set\_power(call\_t \*to, call\_t \*from);

* **transceiver\_set\_power** is used to set the transceiver tx power. Its parameter is the new transmission power in dBm.

void transceiver\_set\_power(call\_t \*to, call\_t \*from, double power);

* **transceiver\_get\_channel** is used to get the transceiver channel. It returns the current transceiver channel.

int transceiver\_get\_channel(call\_t \*to, call\_t \*from);

* **transceiver\_set\_channel** is used to set the transceiver channel. Its parameter is the new transceiver channel.

void transceiver\_set\_channel(call\_t \*to, call\_t \*from, int channel);

* **transceiver\_get\_modulation** is used to get the transceiver modulation. It returns the current transceiver modulation.

classid\_t transceiver\_get\_modulation(call\_t \*to, call\_t \*from);

* **transceiver\_set\_modulation** is used to set the transceiver modulation. Its parameter is the new transceiver modulation.

void transceiver\_set\_modulation(call\_t \*to, call\_t \*from, classid\_t modulation);

* **transceiver\_get\_Tb** is used to get the transceiver bandwidth. It returns the time to transmit a bit in ns.

uint64\_t transceiver\_set\_Tb(call\_t \*to, call\_t \*from);

* **transceiver\_set\_Ts** is used to set the transceiver bandwidth. Its parameter is the new duration of one symbol in ns.

uint64\_t transceiver\_set\_Ts(call\_t \*to, call\_t \*from, uint64\_t Ts);

* **transceiver\_get\_Ts** is used to get the transceiver bandwidth. It returns the time to transmit a symbol in ns.

uint64\_t transceiver\_get\_Ts(call\_t \*to, call\_t \*from);

* **transceiver\_get\_modulation\_bit\_per\_symbol** is used to get the number of bit per symbol for modulation associated. It returns the current number of bit per symbol.

int transceiver\_get\_modulation\_bit\_per\_symbol(call\_t \*to, call\_t \*from);

* **transceiver\_get\_sensitivity** is used to get the transceiver sensibility. It returns the current transceiver sensibility in dBm.

double transceiver\_get\_sensibility(call\_t \*to, call\_t \*from);

* **transceiver\_set\_sensitivity** is used to set the transceiver sensibility. Its parameter is the new transceiver sensibility in dBm.

void transceiver\_set\_sensibility(call\_t \*to, call\_t \*from, double sensibility);

* **transceiver\_cs** is used to notify the transceiver with a new signal. Its parameter is the new packet.

void transceiver\_cs(call\_t \*to, call\_t \*from, packet\_t \*packet);

* **transceiver\_sleep** is used to set the transceiver in sleep mode.

void transceiver\_sleep(call\_t \*to, call\_t \*from);

* **transceiver\_wakeup** is used to set the transceiver in active mode.

void transceiver\_wakeup(call\_t \*to, call\_t \*from);

* **transceiver\_switch\_rx** is used to set the transceiver in RX mode.

void transceiver\_switch\_rx(call\_t \*to, call\_t \*from);

* **transceiver\_switch\_idle** is used to set the transceiver in IDLE mode.

void transceiver\_switch\_idle(call\_ t \*to, call\_t \*from);

These functions are mainly used by MAC layer controlling the radio transceiver.

As an example, we can manage the transceiver at the MAC layer with the following code.

In the MAC bootstrap

call\_t to0 = {get\_class\_bindings\_down(to)->elts[0], to->object};

call\_t from0 = {to->class, to->object};

//wake up the transceiver

transceiver\_wakeup(&to0, &from0);

//start to listen the channel

transceiver\_switch\_rx(&to0, &from0);

The channel is busy if transceiver\_get\_noise(&to0, &from0) >= nodedata->EDThreshold or transceiver\_get\_cs(&to0, &from0) >= nodedata->EDThreshold)

transceiver\_switch\_rx(&to0, &from0) or transceiver\_sleep(&to0, &from0) could be used after a reception or a transmission.

##### Node Mobility

In kernel/model\_handlers/node\_mobility, some declarations are used to link mobility methods:

typedef struct \_mobility\_methods {

void (\*update\_position) (call\_t \*to, call\_t \*from);

double (\*get\_speed) (call\_t \*to);

angle\_t (\*get\_angle) (call\_t \*to);

} mobility\_methods\_t;

The declaration usable by any models are the following:

* **mobility\_get\_angle** is used to get the current angle/direction of the node. It returns a 3D angle.

angle\_t mobility\_get\_angle(call\_t \*to);

* **mobility\_get\_speed** is used to get the current speed of the node.

double mobility\_get\_speed(call\_t \*to);

* **mobility\_get\_mutual\_orientation** is used to get the mutual orientation between two nodes.

double mobility\_get\_mutual\_orientation(call\_t \*mobility\_src, call\_t \*mobility\_dst);

There is two ways to update the position of the node. The first one is to directly add the mobility event in the callback update\_position with the following function:

scheduler\_add\_mobility(get\_time()+MOBILITY\_UPDATE);

With this approach we need to initiate the first event in bind or bootstrap of the mobility model

The second approach is to use mobility\_update(uint64\_t clock) function to update the position

##### CXX\_model\_handlers

Any new C++ model should include this file (kernel/include/model\_handlers/ cxx\_model\_handlers.h) and **implement at least** the **create\_object** and **destroy\_object** functions detailed below. Developers can use the defined macros TO\_C and TO\_CPP to help them to toggle the access of C++ objects between C and C++.

// Use this define to cast the pointer to the object (obj) of type (class) cla.

#define TO\_CPP(obj,cla) (reinterpret\_cast<cla\*>(obj))

// Use this define get the pointer from a C++ object(obj)

#define TO\_C(obj) (reinterpret\_cast<void\*>(obj))

/\*\*

\* \brief The create-object interface, responsible for creating an object of the class and is called by WSNET simulation core during simulation configurations

\* \fn void \*create\_object(call\_t \*to, void \*params)

\* \param to is a pointer to the called class for the given node

\* \param params is a pointer to the parameters read from the XML configuration file

\* \return pointer to the newly created object

\*\*/

void \*create\_object(call\_t \*to, void \*params);

/\*\*

\* \brief The destroy-object interface, responsible for destroying an object and is called by WSNET simulation core at the end of the simulation (cleaning period)

\* \fn void destroy\_object(void \*object)

\* \param object is the pointer to the object to be destroyed

\*\*/

void destroy\_object(void \*object);

/\*\* \brief The bootstrap input-interface which is automatically called by the WSNet simulation core at the beginning of the simulation.

\* \fn int bootstrap(call\_t \*to)

\* \param to is a pointer to the called class for the given node

\* \return SUCCESSFUL if success, UNSUCCESSFUL otherwise

\*\*/

int bootstrap(call\_t \*to);

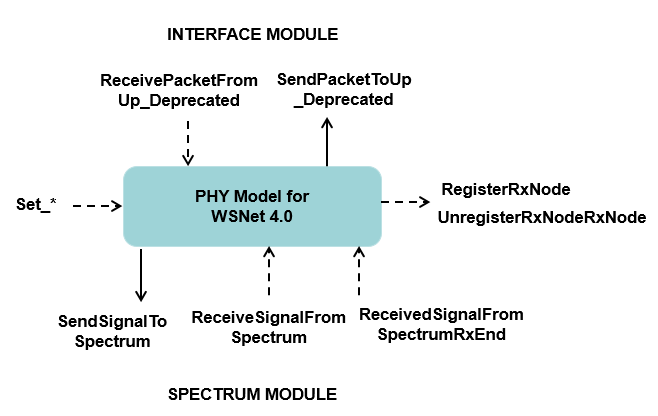
#### Models definitions

In kernel/definitions/models folder, we define abstract base classes of WNET 4.0 medium. This includes a PHY (to Medium) model for each device (including PHY, Coding, Error, Signal Tracker and Interference) and a spectrum model.

##### PHY

In kernel/definitions/models/phy, the abstract base class of PHY model have been defined and linked to Implemented function.

The PHY model API can be defined as follows.



***Figure 41:******PHY Model API.***

The abstract base class of PHY model is:

/\*\* \brief The Abstract Base Class : PhyModel Class

\* This means that no instance of the PhyModel class can exist.

\* Only classes which inherit from the PhyModel class can exist.

\* This model is used to connect nodes to the spectrum

\* The relation with spectrums are:

\* Spectrum has 1..N PhyModels connected to it

\* PhyModel has only 1 Spectrum on which it is connected

\*

\* \fn ReceiveSignalFromSpectrum is used to receive a signal from the spectrum

\* \fn SendSignalToSpectrum is used to send a signal to the spectrum

\* \fn RegisterRxNode is used to register a node on the Rx list of the spectrum

\* \fn UnregisterRxNode is used to remove a node of the Rx list of the spectrum

\* \fn ReceiveSignalFromSpectrumImpl implements the reception of a signal from the spectrum

\* \fn SendSignalToSpectrum implements the transmission of a signal to the spectrum

\* \fn RegisterRxNode implements how to register the node on the Rx list of the spectrum

\* \fn UnregisterRxNode implements how to remove the node of the Rx list of the spectrum

\* \fn ReceivedSignalFromSpectrumRxEnd is to be called on Rx End of the signal

\*\*/

class PhyModel : public std::enable\_shared\_from\_this<PhyModel>{

**public**:

PhyModel();

virtual ~PhyModel();

void **ReceiveSignalFromSpectrum**(std::shared\_ptr<Signal> signal);

void **ReceivePacketFromUp\_Deprecated**(call\_t \*to, call\_t \*from,packet\_t \*packet);

void **SendPacketToUp\_Deprecated**(std::shared\_ptr<Signal> signal);

void **SendSignalToSpectrum**(std::shared\_ptr<Signal> signal);

void **RegisterRxNode**(SetOfFrequencyIntervals freq\_intervals);

void **UnregisterRxNode**();

void **ReceivedSignalFromSpectrumRxEnd**(std::shared\_ptr<Signal> signal);

void **SetInterferenceModel**(InterferenceModel \*);

void **SetModulatorModel**(ModulatorModel \*);

void **SetErrorModel**(ErrorModel \*);

void **SetCodingModel**(CodingModel \*);

void **SetSignalTrackerModel**(SignalTrackerModel \*);

void **SetSpectrum**(SpectrumModel \*);

void Print() const;

PhyModellUid GetUID() const;

**private**:

virtual void ReceiveSignalFromSpectrumImpl(std::shared\_ptr<Signal>)= 0;

virtual void ReceivePacketFromUp\_DeprecatedImpl(call\_t \*to, call\_t \*from,packet\_t \*) = 0;

virtual void SendPacketToUp\_DeprecatedImpl(std::shared\_ptr<Signal>) = 0;

virtual void SendSignalToSpectrumImpl(std::shared\_ptr<Signal>)= 0;

virtual void RegisterRxNodeImpl(SetOfFrequencyIntervals) = 0;

virtual void UnregisterRxNodeRxNodeImpl() = 0;

virtual void ReceivedSignalFromSpectrumRxEndImpl(std::shared\_ptr<Signal> )=0;

virtual void PrintImpl() const;

static PhyModellUid uid\_counter\_;

PhyModellUid uid\_;

**protected**:

InterferenceModel \* interference\_model\_;

ModulatorModel \* modulator\_model\_;

ErrorModel \* error\_model\_;

CodingModel \* coding\_model\_;

SignalTrackerModel \* signal\_tracker\_model\_; // signal tracker is here temporarily. It should be on radio. Radio does not receive signals yet.

MapOfSignals received\_signals\_; // a map containing all received signal being treated at the moment

SpectrumModel \* spectrum\_; // The spectrum on which it is connected

std::shared\_ptr<RegisteredRxNode> registered\_rx\_node\_;

};

In the current version of WSNet 4.0, there is no model handler between the transceiver and the PHY model. Thus, in this first version, we have defined some primitives between the transceiver and the PHY model.

One example is for the RX register functionalities.

In Transceiver side, we have temporarily defined a dummy packet with a primitive as follows:

// This is temporarily being done with dummy packets for now.

// There should be a proper interface between radio and phy to provide

// the exchange of primitives

void register\_on\_spectrum(call\_t \*to){

array\_t \*down = get\_class\_bindings\_down(to);

int i = down->size;

while (i--) {

packet\_t \*packet\_primitive = packet\_create(to,-1,-1);

set\_primitive((char\*) "PHY-SET-REGISTER-RX.request", packet\_primitive);

call\_t from0 = {down->elts[i], to->object};

TX(&from0, to, packet\_primitive);

}

}

In PHY side, when the dummy packet with the primitive is received by ReceivePacketFromUp\_Deprecated, the PHY model can call the associated function (we have temporarily defined a dummy packet with a primitive as follows:

// This is temporarily being done here.

// There should be a proper interface between radio and phy to provide

// the exchange of primitives

if (primitive\_is\_set((char\*)"PHY-SET-REGISTER-RX.request",packet)){

// register rx

RegisterRxNode(frequency\_intervals\_rx\_);

packet\_dealloc(packet);

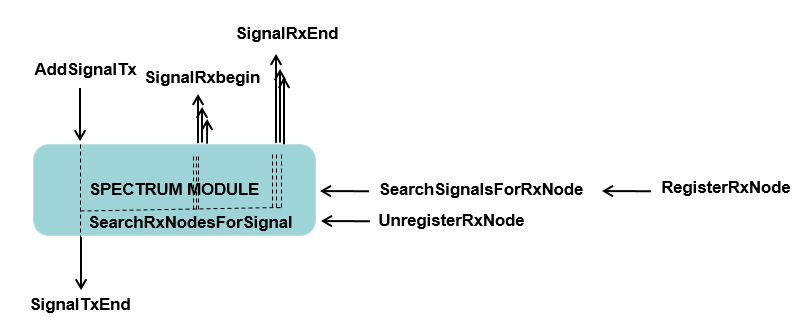
return;

}

##### Spectrum

In kernel/definitions/models/Spectrum, the abstract base class of Spectrum model have been defined and linked to Implemented function.

The Spectrum model API can be defined as follows.



***Figure 42:******Spectrum Model API.***

The abstract base class of spectrum model is:

/\*\* \brief The Abstract Base Class : SpectrumModel Class

This means that no instance of the SpectrumModel class can exist. Only classes which inherit from the SpectrumModel class can exist

The relation with PHY models are: Spectrum has 1..N PhyModels connected to it whereas PhyModel has only 1 Spectrum on which it is connected.

\* \fn UnregisterRXNode is used to unregister a node on the spectrum

\* \fn AddSignalTx is used to notify the spectrum a signal will be transmitted

\* \fn SignalRxBegin is called by the scheduler when a given signal will start to be received

\* \fn SignalRxEnd is called by the scheduler when the reception of a given signal is at the end

\* \fn SignalTxEnd is called by the scheduler when the transmittion of a given signal is at the end

\*\*/

class SpectrumModel{

**public**:

SpectrumModel();

virtual ~SpectrumModel();

std::vector<std::shared\_ptr<Signal>> **RegisterRXNode**(std::weak\_ptr<RegisteredRxNode> rx\_node);

void **UnregisterRXNode**(std::weak\_ptr<RegisteredRxNode> rx\_node);

void **AddSignalTx**(std::shared\_ptr<Signal> signal);

void **SignalRxBegin**(std::shared\_ptr<Signal> signal);

void **SignalTxEnd**(std::weak\_ptr<Signal> signal);

void **SignalRxEnd**(std::shared\_ptr<Signal> signal);

SpectrumUid GetUID() const;

**private**:

//virtual RxNodeFilter GetRegisteredNodeByIdImpl(nodeid\_t) = 0;

virtual std::vector<std::shared\_ptr<Signal>> RegisterRxNodeImpl(std::weak\_ptr<RegisteredRxNode>)=0;

virtual void UnregisterRxNodeImpl(std::weak\_ptr<RegisteredRxNode>)=0;

virtual void SignalAddTxImpl(std::shared\_ptr<Signal>) = 0;

virtual void SignalRxBeginImpl(std::shared\_ptr<Signal>) = 0;

virtual void SignalTxEndImpl(std::weak\_ptr<Signal> signal) = 0;

virtual void SearchRxNodesForSignalImpl(std::weak\_ptr<Signal>) = 0;

virtual std::vector<std::shared\_ptr<Signal>> SearchSignalsForRxNodeImpl(std::weak\_ptr<RegisteredRxNode>) = 0;

virtual void SignalRxEndImpl(std::shared\_ptr<Signal>) = 0;

**protected:**

void SearchRxNodesForSignal(std::weak\_ptr<Signal> signal);

std::vector<std::shared\_ptr<Signal>> SearchSignalsForRxNode(std::weak\_ptr<RegisteredRxNode> rx\_node);

static SpectrumUid uid\_counter\_;

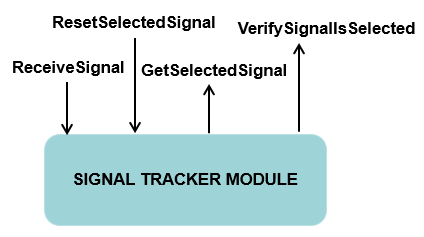
SpectrumUid uid\_;

};

##### Signal Tracker

In kernel/definitions/models/Signal Tracker, the abstract base class of Signal Tracker model have been defined and linked to Implemented function. The signal tracker model is responsible for the capture effect of the transceiver i.e. on which signal the transceiver decide to start the decoding.

The Signal Tracker model API can be defined as follows.



***Figure 43:******Signal Tracker Model API.***

The abstract base class of Signal Tracker model is:

/\*\* \brief The Abstract Base Class : SignalTrackerModel Class

\* This means that no instance of the Interval class can exist.

\* Only classes which inherit from the Interval class can exist.

\*

\* \fn GetSelectedSignal() return the selected signal

\* \fn ResetSelectedSignal() resets the selected signal

\* \fn VerifySignalIsSelected() checks if the signal is selected or not

\* \fn ReceiveSignal() receive a new signal

\* \fn Print() prints information about the SignalTrackerModel

\* \fn GetUID() return the UID of the SignalTrackerModel

\* \fn VerifySignalIsSelectedImpl() implements the method that checks if the signal is selected or not (to be implemented by derived class)

\* \fn GetSelectedSignalImpl() implements the function that get the selected the signal that will be received (to be implemented by derived class)

\* \fn ResetSelectedSignalImpl() implements the function that resets the selected the signal (to be implemented by derived class)

\* \fn ReceiveSignalImpl() implements the function that selects the signal that will be received (to be implemented by derived class)

\* \fn PrintImpl() implemetns the Print function (to be implemented by derived class)

\*\*/

class SignalTrackerModel {

public:

SignalTrackerModel();

virtual ~SignalTrackerModel();

std::shared\_ptr<Signal> **GetSelectedSignal**();

bool **VerifySignalIsSelected**(std::shared\_ptr<Signal> signal);

bool **ReceiveSignal**(std::shared\_ptr<Signal> signal);

void **ResetSelectedSignal**();

void Print() const;

SignalTrackerModelUid GetUID() const;

private:

virtual std::shared\_ptr<Signal> GetSelectedSignalImpl() =0 ;

virtual void ResetSelectedSignalImpl() = 0;

virtual void PrintImpl() const;

virtual bool ReceiveSignalImpl(std::shared\_ptr<Signal>) = 0;

virtual bool VerifySignalIsSelectedImpl(std::shared\_ptr<Signal>) = 0;

static SignalTrackerModelUid uid\_counter\_;

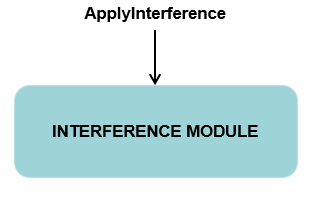
SignalTrackerModelUid uid\_;

};

##### Interference

In kernel/definitions/models/Interference, the abstract base class of Interference model have been defined and linked to Implemented function. The Interference model is the interference model for signals.

The Interference model API can be defined as follows.



***Figure 44:******Interference Model API.***

The abstract base class of Interference model is:

/\*\* \brief The Abstract Base Class : InterferenceModel Class

\* This means that no instance of the InterferenceModel class can exist.

\* Only classes which inherit from the InterferenceModel class can exist.

\*

\* \fn ApplyInterference() apply the interferences of interferers on the tracked signall

\* \fn Print() prints information about the InterferenceModel

\* \fn GetUID() return the UID of the InterferenceModel

\* \fn ApplyInterferenceImpl() implements the application of interferences on the tracked signal

\* \fn PrintImpl() implements the Print function

\*\*/

class InterferenceModel {

public:

InterferenceModel();

virtual ~InterferenceModel();

void **ApplyInterference**(std::shared\_ptr<Signal> tracked\_signal, MapOfSignals interferers);

void Print() const;

InterferenceModelUid GetUID() const;

private:

virtual void ApplyInterferenceImpl(std::shared\_ptr<Signal> , MapOfSignals) =0 ;

virtual void PrintImpl() const;

static InterferenceModelUid uid\_counter\_;

InterferenceModelUid uid\_;

};

##### Coding and Error

In kernel/definitions/models/coding and kernel/definitions/models/error, the abstract base class of coding and error models have been defined and linked to Implemented function. These models are not used yet.

The abstract base class of coding model is:

/\*\* \brief The Abstract Base Class : CodingModel Class

\* This means that no instance of the CodingModel class can exist.

\* Only classes which inherit from the CodingModel class can exist.

\*

\* \fn ApplyCoding() apply the coding on the signal

\* \fn Print() prints information about the InterferenceModel

\* \fn GetUID() return the UID of the InterferenceModel

\* \fn ApplyCodingImpl() implements the application of coding on the signal

\* \fn PrintImpl() implemetns the Print function

\*\*/

class CodingModel {

public:

CodingModel();

virtual ~CodingModel();

void **ApplyCoding**(std::shared\_ptr<Signal> signal);

void Print() const;

CodingModelUid GetUID() const;

private:

virtual void ApplyCodingImpl(std::shared\_ptr<Signal>) =0 ;

virtual void PrintImpl() const;

static CodingModelUid uid\_counter\_;

CodingModelUid uid\_;

};

The abstract base class of error model is:

/\*\* \brief The Abstract Base Class : ErrorModel Class

\* This means that no instance of the ErrorModel class can exist.

\* Only classes which inherit from the ErrorModel class can exist.

\*

\* \fn ApplyErrors() apply the errors on the signal

\* \fn Print() prints information about the ErrorModel

\* \fn GetUID() return the UID of the ErrorModel

\* \fn ApplyErrorsImpl() implements the application of errors in the signal (to be implemented by derived class)

\* \fn PrintImpl() implemetns the Print function (to be implemented by derived class)

\*\*/

class ErrorModel {

public:

ErrorModel();

virtual ~ErrorModel();

void **ApplyErrors**(std::weak\_ptr<Signal> signal);

void Print() const;

ErrorModelUid GetUID() const;

private:

virtual void ApplyErrorsImpl(std::weak\_ptr<Signal>) =0 ;

virtual void PrintImpl() const;

static ErrorModelUid uid\_counter\_;

ErrorModelUid uid\_;

};

#### Other functions

In kernel/math\_toolbox/rng.h&c, some random generators are defined:

* **get\_random\_distance** returns a random distance

double get\_random\_distance(void);

* **get\_random\_x\_position/ get\_random\_y\_position / get\_random\_z\_position** return a random position on the x/y/z dimension

double get\_random\_x\_position(void);

double get\_random\_y\_position(void);

double get\_random\_z\_position(void);

* **get\_random\_double** returns a random double value in [0,1]

double get\_random\_double(void);

* **get\_random\_double\_range** returns a random double value in [min,max]

double get\_random\_double\_range(double min, double max);

* **get\_random\_integer** returns a random integer value

int get\_random\_integer(void);

* **get\_random\_integer\_range** returns a random integer value in [min,max]

int get\_random\_integer\_range(int min, int max);

* **get\_random\_time** returns a random time in [0, simulation\_end]

uint64\_t get\_random\_time(void);

* **get\_random\_time\_range** returns a random time in [min,max]

uint64\_t get\_random\_time\_range(uint64\_t min, uint64\_t max);