Cohabitation between the encoding functions of Protomatics and the modeling tool of Pragmadev

Internship 2014
ANDRES Elie
ESIEE Paris

Table of Contents

Introduction :

This documentation is made to explain how to use the software of Protomatics with the tool of Pragmadev, RTDS.

Protomatics proposes a compiler generating some data types and functions allowing the user to encode and decode a message.

RTDS, used in SDL-RT is made for the modeling and the code generation of a protocol.

The purpose of the project is to show the possibility of cohabitation between the two software.

This cohabitation can be difficult under the OS windows, due to some complication with the command simulator Cygwin.

This project has been reached under Linux.

First, we define the data with the TSN.1 notation, and generate the code corresponding.

Then we will see the use of this generating code with the code C.

Finally, we will see how we included in a protocol the encoding and decoding functions of Protomatics.

The TSN.1 notation and the TSN.1 Compiler:

Definition:

The TSN.1 or Transfer Syntax Notation One allows the user to define some data type included in a message.

The big advantage of this notation compared to the ASN.1 (Abstract Syntax Notation) is to describe the data in a binary representation.

The ASN.1 is using abstract types to define his data (integer ...) but don't specify automatically the specific rule needed to encode the data, this means transforming the definition into concrete bits. (For different type of data, several rules of encoding can be employed).

Instead, TSN.1 gives us two information with a single notation:

- The appropriate information bits. (the bits of the abstract type to take)
- The number of encoding bits. (the number of bits to encode)

Hence, the message defines directly in binary can be encoded automatically with a single rule.

TSN.1 comports conditionals, and is made to be intuitive. The purpose is make the encoding more flexible and customisable.

The software of Protomatics can be download on this link:

http://www.protomatics.com/trial.html

All documentations can be found on this web page:

http://www.protomatics.com/support.html

Code generation:

When a file is written in tsn.1, the text file must be named with the extension ".tsn".

The purpose of the TSN.1 Compiler is to generate two files from the one we created in tsn.1. These two files correspond to a header and a file in C or C++ code, depending of the type of code we want to have.

Several functions are available, the main ones are named pack and unpack. They allows the user to encode and decode the file describe in tsn.1.

In order to generate the new files, we open a terminal, go into the binary directory of Protomatics and enter the command:

tsnc -pack -unpack -d /generated_code_directory_path/ /path_file/name_file.tsn

-d is an option, it allows the user to choose the directory inside which source and header files will be generated.

Here, we will by default generate a C file. If we want to generate C++, we add -cxx after tsnc.

The generated C or C++ files must not be modified. They are used only by the compiler runtime Library.

Description of TSN.1:

When we used the notation TSN.1, we can see the corresponding types generated in the header file. Here, we will see all the different configurations that are generated for our messages.

TSN.1 notation:	Generated header:

wimax_mac_PDU()::=
{
 	
header_Mac 48 :
{
HT 1; /* header type */
EC 1; /* encryption control */
if(HT == 1)
{
Type 6; /* type */
reserve 1; /* reserved */
CI 1; /* CRC indicator */
EKS 2; /* Encryption Key sentence */
reserve 1;
LEN 11; /* Length of the mac_PDU (including header and CRC) */
}
else
{
Type 3 (0 .. 1); // type
BR 19; //Bandwidth request
}
CID 16; /* Connexion Identifier */
HCS 8; /* Header Check Sequence */
}
 	
Management_Message_Type 8;
 	
Value : case Management_Message_Type of
{
0 =>
UCD : UCD_Message_Format;
1 =>
DCD : DCD_Message_Format;
2 =>
DL_MAP : DL_MAP_Message_Format;
3 =>
UL_MAP : UL_MAP_Message_Format;
[...]
}
}
	TSNC_EXTERN tsnc_msg_Descriptor wimax_mac_PDU_header_Mac_descriptor;
	
	typedef struct _wimax_mac_PDU_header_Mac
	{
	tsnc_Message __tsnc_msg__;
	
	/* Fields */
	tsnc_uint8 HT;
	tsnc_uint8 EC;
	tsnc_uint8 Type;
	tsnc_uint8 CI;
	tsnc_uint8 EKS;
	tsnc_uint16 LEN;
	tsnc_uint32 BR;
	tsnc_uint16 CID;
	tsnc_uint8 HCS;
	
	/* DO NOT USE!!! Internal fields */
	struct _wimax_mac_PDU *__parent__;
	} wimax_mac_PDU_header_Mac;
	
	TSNC_EXTERN tsnc_msg_Descriptor wimax_mac_PDU_descriptor;
	
	typedef struct _wimax_mac_PDU
	{
	tsnc_Message __tsnc_msg__;
	
	/* Fields */
	wimax_mac_PDU_header_Mac *header_Mac;
	tsnc_uint8 Management_Message_Type;
	
	union
	{
	UCD_Message_Format *UCD;
	DCD_Message_Format *DCD;
	DL_MAP_Message_Format *DL_MAP;
	UL_MAP_Message_Format *UL_MAP;
	[...]
	} Value;
	
	} wimax_mac_PDU;

We can see that the syntax of TSN.1 is quite simple.

- Here we created one message **wimax_mac_PDU()::={}** with a nested message inside him header_Mac 48: () (48 indicates a fixed size for this message).

It is important to see that every message is defined as a structure. But every nested message will be declared as pointers contained in the previous structure:
wimax_mac_PDU_header_Mac *header_Mac;

- We put the type with the number of bits corresponding to it, it is translated with a basic type such as tsnc_uint8 for instance.

We can also see that some options can be added: Type 3 (0 .. 1); has a delimitation for his value from 0 to 1.

The type defined is unsigned by default, we can add signed at the end the line if we want it to be.

- reserve represent the reserved bits of the message, they are fixed to 0 and don't appears in the header.

- This message **wimax_mac_PDU()::={}** comport a conditional depending on the value of the field **HT**. It is a security for further use of the structure. The types depending on the field **HT** will be chosen according to his value.

- Last but not least, we can see a : Value : case Management_Message_Type of {}

This means that the value of the Management_Message_Type will determined the future message to use.

For instance 0 will be calling the message UCD with type UCD_Message_Format.

Here in this part we will see three important things: an unbounded array, the definition of a macro, and his utilization with a **case ... of**

TSN.1 notation:

```
TLV_Length ::= macro
{
        Length_MSB 1;
        if(Length_MSB == 0)
        {
                Length 7;
        }
        if(Length_MSB == 1)
        {
                Length_Size 7 (0 .. 2);
                Length? Length_Size * 8;
        }
}

[...]

RNG_RSP_Message_Format()::=
{
        reserve 7;
        H_FDD_Group_Indicator 1;

        TLV_Ranging_Info[] :
        {
                Type 8;
                expand TLV_Length;
                TLV_ranging Length*8 : case Type of
                {
                        1 =>
                        Timing_Ajust 32 signed;
                        2 =>
                        Power_Level_Ajust 8 signed;
                        3 =>
                        Offset_Frequency_Ajust 32
signed;
                        4=>
                        Ranging_Status   8;
 [...]
                }

        }
}
```

Generated header:

```
TSNC_EXTERN tsnc_msg_Descriptor
RNG_RSP_Message_Format_TLV_Ranging_Info_descripto
r;

typedef struct
_RNG_RSP_Message_Format_TLV_Ranging_Info
{
  tsnc_Message __tsnc_msg__;

  /* Fields */
  tsnc_uint8   Type;
  tsnc_uint8   Length_MSB;
  tsnc_uint16  Length;
  tsnc_uint8   Length_Size;

  union
  {
    tsnc_int32   Timing_Ajust;
    tsnc_int8    Power_Level_Ajust;
    tsnc_int32   Offset_Frequency_Ajust;
    tsnc_uint8   Ranging_Status;
[...]
  } TLV_ranging;


  /* DO NOT USE!!! Internal fields */
  tsnc_uint8   __Length_present__;
  struct _RNG_RSP_Message_Format *__parent__;
} RNG_RSP_Message_Format_TLV_Ranging_Info;

TSNC_EXTERN tsnc_msg_Descriptor
RNG_RSP_Message_Format_descriptor;

typedef struct _RNG_RSP_Message_Format
{
  tsnc_Message __tsnc_msg__;
  /* Fields */
  tsnc_uint8   H_FDD_Group_Indicator;
  tsnc_uint16  _TLV_Ranging_Info_size_;
  RNG_RSP_Message_Format_TLV_Ranging_Info
*TLV_Ranging_Info[16];

} RNG_RSP_Message_Format;
```

First, lets says that this message hasn't the obligation to fill all the types he had.

Here the message comport some TLV encoding data.

The TLV data are created as follow :

- Type : a number specific to the value

- Length : the length in number of bytes
- Value : the value of the type concerned (always coded as a multiple of a byte).

As a **Case Type of** , the TLVs are defined in an union. So only one TLV can be assigned at a time.

We don't know how many TLV there will be when we will send the message , so the unbounded array TLV_Ranging_Info[] : generates in the header a _TLV_Ranging_Info_size_ which indicate the size of the array, so the number of TLV we want according to our need.

The generation creates an array of structure's pointers.

(By default the size of the array is limited to 16, but can be fixed to an higher value)

If the size of the array is fixed, ie : foo[6], the arrays will always have 6 parameters inside.

The macro here allows the Length of the TLV data.

Described in TSN.1 the macro is there to see if 7 bits are enough to describe the length in bytes of the TLV. The Length_MSB; indicates if one byte is enough (= 0), if not, the size of Length become two bytes. Then we could fix him a value superior at 127.

Length = Length_size * 8. and the value of Length_Size is at maximum 2.

So every time a TLV need to be occupied we need at least to take care of the fields Length_MSB and Length.

Summary :

- “The first Message” containing the other ones is defined as a structure.
- “Nested Messages” are defined as pointers of a structure.
- “Bound or Unbounded Arrays” are defined an Array of pointers of structures.
- “Case of” are generated as an union. On which we have to fix the length for TLV data.

C code under RTDS

The needed files

Some files need to be include or link under the project in order to use the new data types and the Protomatics functions.

- First we have to include the two files generated.

The header is giving the definitions of the different types.

The C generated file is automatically called by the functions of Protomatics, the user must not touch it.

- Then we have to add the headers given in the repertory /src in Protomatics:

- tsnc.h : defines the different error messages that we can have.
- tsnc_buf.h : defines the functions to pack/unpack integers into bit.
- tsnc_custom.h : defines the the new basic types to use. Example : tsnc_uint16.
- tsnc_msg.h : defines the macro of the different functions.
- tsnc_rte.h : defines the common definitions used by the functions.

If we want to compile with g++ instead of gcc, (changing in the options of compilation in RTDS)

We need to had tsncxx_msg.h to the list and remove tsnc_msg.h.

(This solution hasn't been tested).

- Finally, the static library containing the different functions of Protomatics into the compilations options of RTDS as a file to link. This library depends of the OS. We found her in the repertory /lib.

Description of the messages

Definition of a message

In the model, all the different types of messages used in RTDS come from the same basic structure : wimax_mac_PDU.

First structure in the header containing all the types:

```
typedef struct _wimax_mac_PDU
{
tsnc_Message __tsnc_msg__;
/* Fields */
wimax_mac_PDU_header_Mac *header_Mac;
tsnc_uint8 Management_Message_Type;
union
{
UCD_Message_Format *UCD;
DCD_Message_Format *DCD;
DL_MAP_Message_Format *DL_MAP;
UL_MAP_Message_Format *UL_MAP;
RNG_REQ_Message_Format *RNG_REQ;
RNG_RSP_Message_Format *RNG_RSP;
} Value;
} wimax_mac_PDU;
```

Here we have 6 different types of messages in our system: UCD, DCD, DL_MAP, UL_MAP, RNG_REQ, RNG_RSP, all defined in the same structure.

If we want two messages present at the same time, we will have to allocate twice the memory for the structure wimax_mac_PDU.

Architecture of a message

This example shows the different parts of a message from the top structure to the final one.

The data types have been removed from the structures and the unions.

DCD : Downlink channel descriptor message

```
typedef struct _wimax_mac_PDU
{
union { DCD_Message_Format *DCD; } Value;
} wimax_mac_PDU;


typedef struct _DCD_Message_Format
{
tsnc_uint16 _TLV_Channel_DCD_size_;
DCD_Message_Format_TLV_Channel_DCD *TLV_Channel_DCD[16];
} DCD_Message_Format;
```

```
typedef struct _DCD_Message_Format_TLV_Channel_DCD
{
_NBR_Burst_Profile_Downlink_size_;
union { Burst_Profile_ Downlink *NBR_Burst_Profile_Downlink[3]; } Value;
} UCD_Message_Format_TLV_Channel_UCD;


typedef struct _Burst_Profile_ Downlink
{
union { } Value;
} Burst_Profile_Downlink;
```

The architecture of the DCD message is important in order to fill the corrects data types that are included in the different levels. That shows the paths we have to follow in order to access a data.

Here we search to have the most distant data from the wimax_mac_PDU structure:

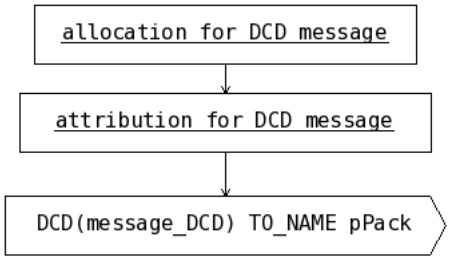
wimax_mac_PDU. DCD-> TLV_Channel_DCD[]->Value. NBR_Burst_Profile_Downlink[]->Value.Data

Using the data types

A message must follow some steps before we can send him:

- The declaration of a pointer as a local variable with type "wimax_mac_PDU"
- The allocation of the memory to use.
- The assignation of value to the differents data types.

Here, the "allocation for DCD message" and "attribution for DCD messages" are task block. They are fill with C code, as we will see later.



At last, the message is prepared to be encoded, and can be send.

Allocation of memory

First of all we have to declare a pointer on the principal structure that contains all the others messages as a local variable.

In our case the structure is wimax_mac_PDU. Hence the pointer will be personalized with the message.

Here, we took for the example the DCD_message:

```
wimax_mac_PDU* message_DCD;
```

Dynamic allocation :

Then, we have to allocate some new memory for the pointers contained in the structure.

To do so, we use the function tsnc_message_new(Structure_Type).

The function has two advantages, she dynamically allows the memory of the type given, and returns a pointer that we can stock.

Example of allocation for DCD_message :

```
/* header */
message_DCD->header_Mac = tsnc_msg_new(wimax_mac_PDU_header_Mac);
/* DCD message */
message_DCD->Value.DCD = tsnc_msg_new(DCD_Message_Format);
/*TLV encoded informations */
for(j= 0; j< 4; j++)
{
message_DCD->Value.DCD->TLV_Channel_DCD[j] =
      tsnc_msg_new(DCD_Message_Format_TLV_Channel_DCD);
}
/* Burst Profiles Downlink */
for(j=0; j<2; j++)
{
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[j]=
tsnc_msg_new(Burst_Profile_Downlink);
}
```

For instance, the pointer `header_Mac` pointed by `message_DCD` will correspond to the memory allocated by the function `tsnc_msg_new(wimax_mac_PDU_header_Mac)`. We have allocated the size of the structure corresponding to a header_Mac.

For the unbounded arrays we need to create a loop for the number of structures we want. Here we will have 4 differents `TLV_Channel_DCD`, the first one conaining an union pointing to an array of 2 `NBR_Burst_Profile_Downlink`.

Initialization

We have to allocate the memory needed for the pointer. Here it correspond to the size of the structure wimax_mac_PDU.

```
message_DCD = malloc(sizeof(wimax_mac_PDU));
```

- The new message must be properly initialize before any use is made with him:

```
tsnc_msg_initialize(message_DCD,wimax_mac_PDU);
```

Dynamic allocation

Then, we have to allocate some new memory for the pointers contained in the structure.

To do so, we use the function tsnc_message_new(Structure_Type).

The function has two advantages, she dynamically allows the memory of the type given, and returns a pointer that we can stock.

Example of allocation for DCD_message :

```
/* header */
message_DCD->header_Mac = tsnc_msg_new(wimax_mac_PDU_header_Mac);
/* DCD message */
message_DCD->Value.DCD = tsnc_msg_new(DCD_Message_Format);
/*TLV encoded informations */
for(j= 0; j< 4; j++)
{
message_DCD->Value.DCD->TLV_Channel_DCD[j] =
      tsnc_msg_new(DCD_Message_Format_TLV_Channel_DCD);
}
/* Burst Profiles Downlink */
for(j=0; j<2; j++)
{
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[j]=
tsnc_msg_new(Burst_Profile_Downlink);
}
```

For instance, the pointer `header_Mac` pointed by `message_DCD` will correspond to the memory allocated by the function `tsnc_msg_new(wimax_mac_PDU_header_Mac)`. We have allocated the size of the structure corresponding to a header_Mac.

For the unbounded arrays we need to create a loop for the number of structures we want. Here we will have 4 differents `TLV_Channel_DCD`, the first one conaining an union pointing to an array of 2 `NBR_Burst_Profile_Downlink`.

Attribution for the types

After the memory is allocated it is time to fill the data types that we have created.

```
/* header_MAC */
message_DCD->header_Mac->HT = 1;
message_DCD->header_Mac->EC = 0;
message_DCD->header_Mac->Type = 0;
message_DCD->header_Mac->CI = 0;
message_DCD->header_Mac->EKS = 0;
message_DCD->header_Mac->LEN = sizeof(wimax_mac_PDU) +
sizeof(wimax_mac_PDU_header_Mac) + sizeof(DCD_Message_Format) +
sizeof(DCD_Message_Format_TLV_Channel_DCD)*4 + sizeof(Burst_Profile_Downlink)*2;
message_DCD->header_Mac->CID = 0xFFFF;
message_DCD->header_Mac->HCS = 0;


message_DCD->Management_Message_Type = 1;


/* TLV informations for DCD */
message_DCD->Value.DCD->_TLV_Channel_DCD_size_ = 4;


/* Burst profile DCD informations */
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Type = 1;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Length_MSB = 0;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Length = 8;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->_NBR_Burst_Profile_Downlink_size_ =
2;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[0]-
>Type = 150;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[0]-
>Length_MSB = 0;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[0]-
>Length = 1;
```

```
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[0]->Value.FEC_Code_Type = 6;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[1]->Type = 153;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[1]->Length_MSB = 0;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[1]->Length = 1;
message_DCD->Value.DCD->TLV_Channel_DCD[0]->Value.NBR_Burst_Profile_Downlink[1]->Value.TCS_enable = 0;


message_DCD->Value.DCD->TLV_Channel_DCD[1]->Type = 6;
message_DCD->Value.DCD->TLV_Channel_DCD[1]->Length_MSB = 0;
message_DCD->Value.DCD->TLV_Channel_DCD[1]->Length = 1;
message_DCD->Value.DCD->TLV_Channel_DCD[1]->Value.Channel_Nr = NrChannel;
message_DCD->Value.DCD->TLV_Channel_DCD[2]->Type = 23;
message_DCD->Value.DCD->TLV_Channel_DCD[2]->Length_MSB = 0;
message_DCD->Value.DCD->TLV_Channel_DCD[2]->Length = 1;
message_DCD->Value.DCD->TLV_Channel_DCD[2]->Value.Available_DL_Radio_Ressources = 0xFF;
message_DCD->Value.DCD->TLV_Channel_DCD[3]->Type = 148;
message_DCD->Value.DCD->TLV_Channel_DCD[3]->Length_MSB = 0;
message_DCD->Value.DCD->TLV_Channel_DCD[3]->Length = 1;
message_DCD->Value.DCD->TLV_Channel_DCD[3]->Value.MAC_Version = 0;
```

Here, we have to be careful at two things:

- The arrays. It is mandatory to fill the name_array_size with the value corresponding of the number of structures populated.

- The TLVs data. The TLVs data depend also on others field: the Length and the Length_MSB.

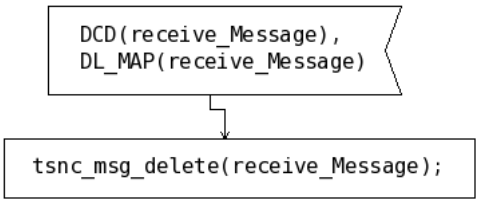
It's important that the Length field (in byte) correspond to the size of the "Value" of the TLV.

Free the memory

When we have finished with the message, we need to free the allocated memory. Again we use a Protomatics's function which free the internal pointers recursively.

tsnc_msg_delete().

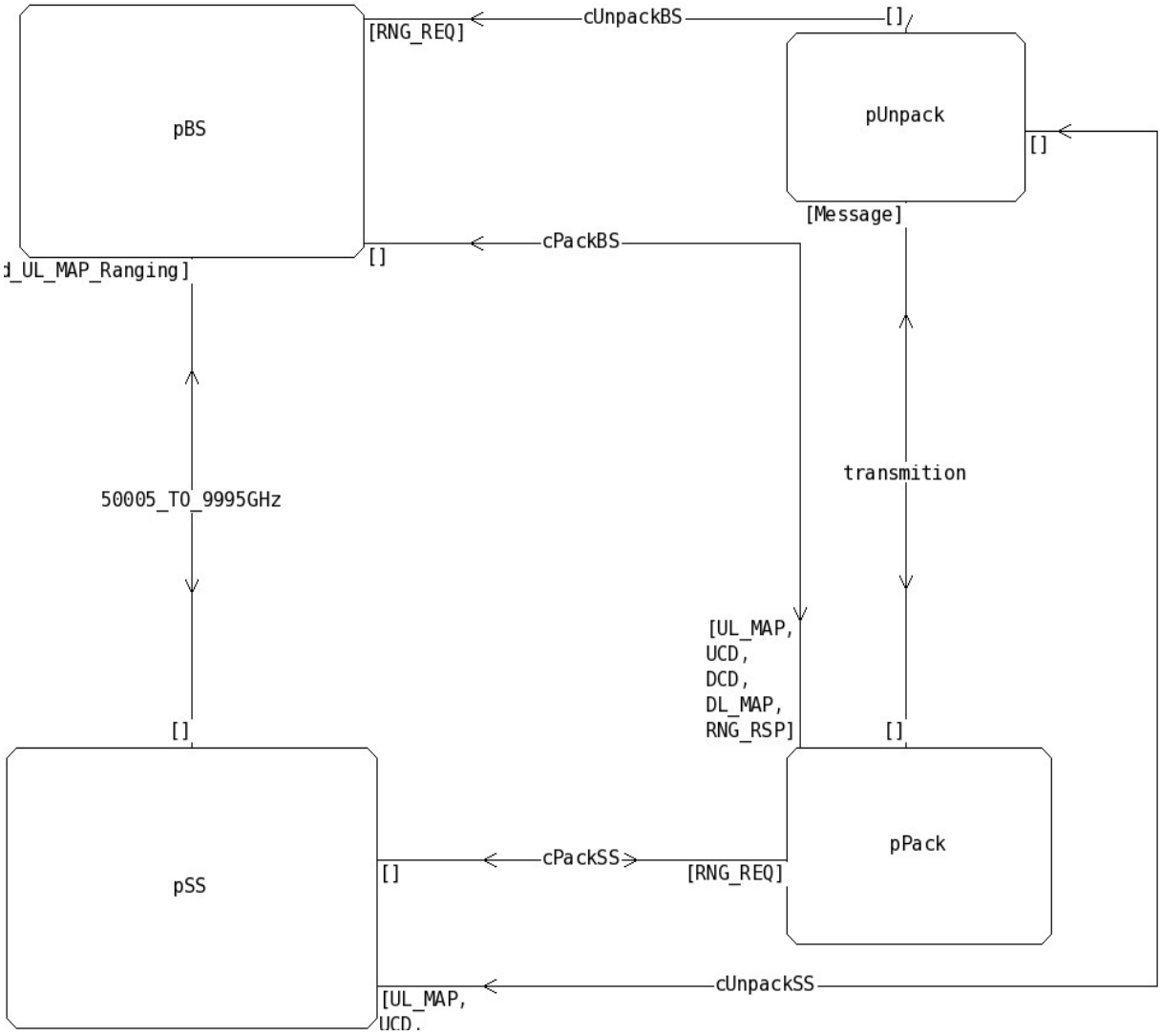
This function take for parameter the pointer of the structure we need to free.



Note that the process responsable of the message is the one receiving it. In this example, if the process receive a DCD or an DL_MAP message he will automatically free the memory of the parameters.

Encoding and deconding under RTDS

Every time a message is sent, we need to show the encoding and the decoding of the PDU. To do so two process are created, the process pPack and pUnpack. They will work one the mesasges transmitted by pSS and pBS.



Here, we can see in this example that all the wimax messages sent are transmitted first to the pPack process, then to the pUnpack one before to be deliver to the interested process.

For instance, the UL_MAP message comming from pBS is transmited to pPack, then into the unknown form of Message to pUnpack. And finnaly received by pSS

From sender to Pack

Even if the Type of the message is declared into the description in tsn.1. We transmit a named message to the function pPack, containing the pointer concerning the message.

Assuming the Sender is also the encoder, he must have knowledge of the message he encodes.

The Pack process

The process pack encode the data into a buffer of bits.

To do so, we send the messages to the process pPack . The process is prepare to receive all kind of messages the communicating processes can exchange.

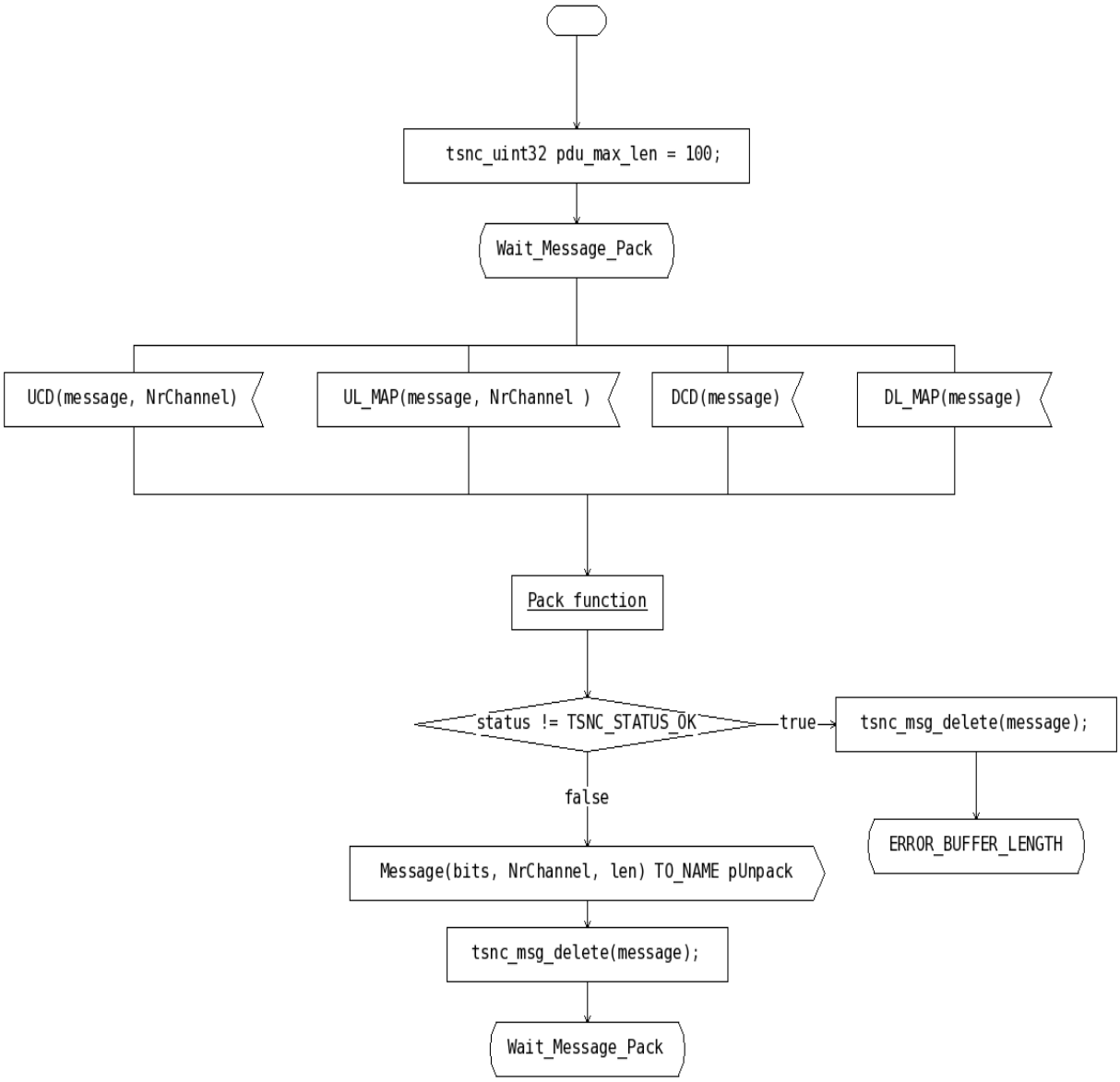
He retrieve the pointer and call the function pack(), this one returns a status:

```
status = tsnc_msg_pack(message, &bits, 0, pdu_max_len * 8, &len);
```

This one takes the following parameters:

- `message` : pointeur of the structure to pack into.
- `&bits` : pointer of the buffer (type tsnc_uint8, length 100).
- `0` : The offset, the number of bits of the buffer it beginns to pack into.
- `pdu_max_len * 8` : the maximum length of the buffer in bits.
- `&len` : returns in the variable "len" the number of bit packed.

At last we convert the size "len" in byte for more clarity: `len >>= 3;`



Here we have a security. If one of the message had a problem to be packed, the function returns a error_status indicating a failure, in the allocation of the memory or in the attribution of the data types, if the buffer to pack into is too small...

Then, he send the buffer as a parameter in an anonymous message with the len corresponding to the number of bytes encoded, the third parameter isn't use by the functions of Protomatics, but need to be sent for the protocol.

It's a choice not to send a pointer of the buffer in order to have it as a parameter, so we can read the bytes directly when generating MSC trace with the debugger of RTDS.

At the end of it, we free the memory with the tsnc_msg_delete(). Here we don't have to be concerned about the messages sent, the pointers are inevitably free of their memory at this stage.

The Unpack process

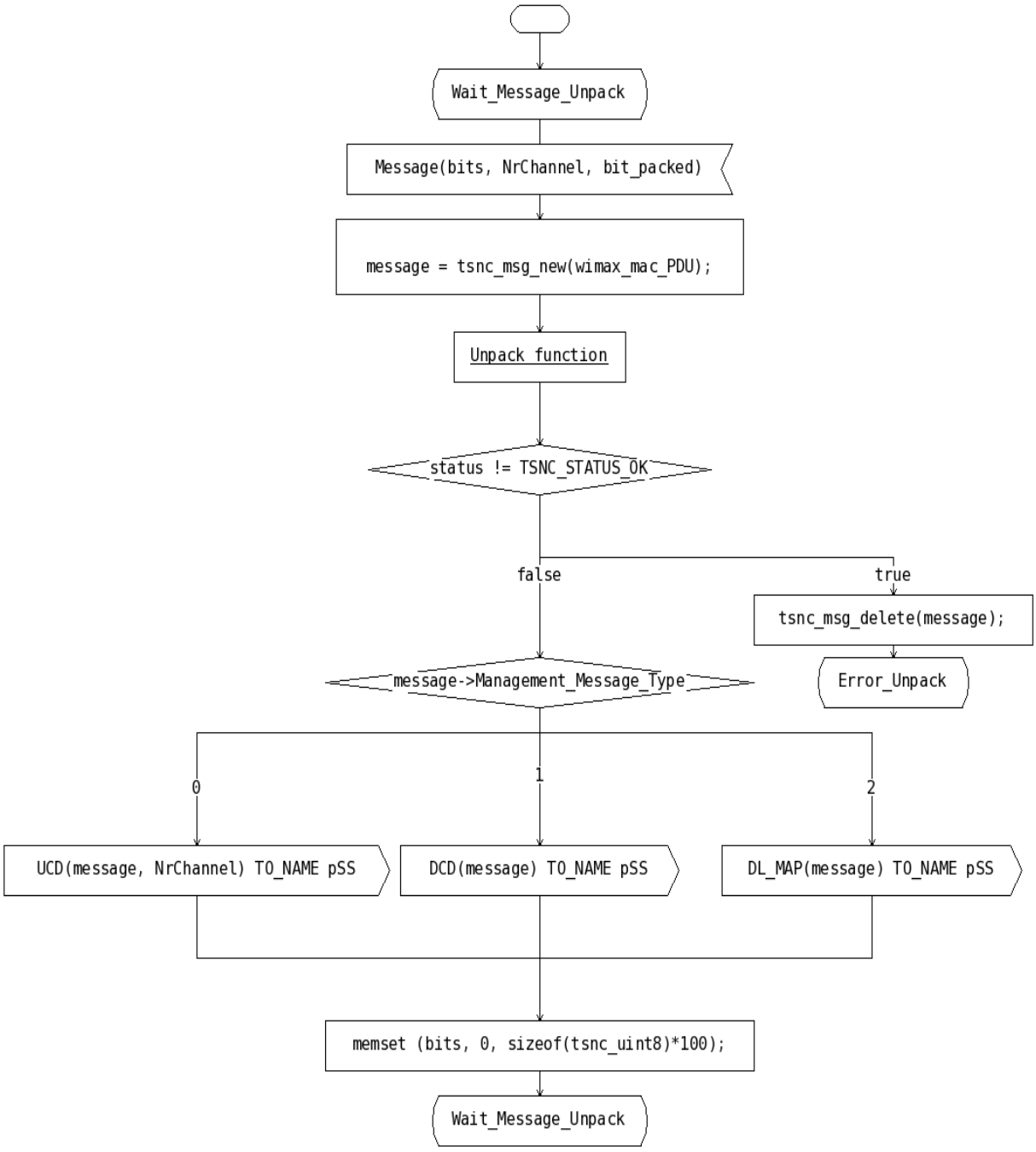
The unpack process receive an unknown message, and collect the buffer and the number of bits encoded.

Here, the tsnc_msg_unpack() function do all the work, she allocate from the transmitted pointer the memory dynamically and fix the different data types needed.

```
status = tsnc_msg_unpack(message, &bits, 0, bit_packed*8, &bit_unpacked);
```

This one takes the following parameters:

- `message` : The message to unpack in. (pointer with type wimax_mac_PDU)
- `&bits` : pointer to the buffer of bits receive. (array of tsnc_uint8, with a length of 100)
- `0` : The offset, the number of bit of the buffer to start unpack for.
- `bit_packed*8` : The number of bit packed. Corresponding to the parameter "len" (in byte) send by pPack.
- `&bit_unpacked` : returns the number of bits unpacked.



Same as for the function pack we check if there is an error while executing the unpack function.

Then, in order to send the good message we check the Management_Message_Type contained. This particular type gives us the name of the message we have to pass by.

Here for reasons of scale we just put three of the messages that pUnpack can forward.

Then we send it with the pointer of the unpacked structure inside.

From Unpack to the receiver

This is the job of the receiver to have prepared a pointer of the same structure wimax_mac_PDU (here named receive_Message), to use the information who needs to be treated, then to free the memory when it has no longer use.

NB: to avoid memory leaks, all the messages sent, even if they are not useful for the receiver, must be free by the process we sent them to.

Conclusion

In definitive, the project shows that the interoperability between RTDS and the functions of Protomatics is duable.

The compilater of RTDS supports the runtime library implemented by Protomatics.

The protomatics functions can be used with the SDL_RT language.

Meanwhile a large part for the allocation of the memory and the attributions of the values are essentially coded in c-language, this part is easilly done with the copy/ paste utilities.

The only tricky part is to watch that all the parameters/pointers are free when their are of no use in order to have no memory leacks.