Implementation Notes for the GSM BTS Model

Sorin Manolache, Razvan Jigorea
ESLAB, Linköping University

1. General Description

The general architecture of the model is depicted in Figure 1. Every block in the figure is mapped to a package in the model, with a single exception: the two protocol interpreters (LAPDm PI and LAPD PI) are mapped to the same package (*PIs*) as they are very similar and result from simple parameterizations of a super-class.

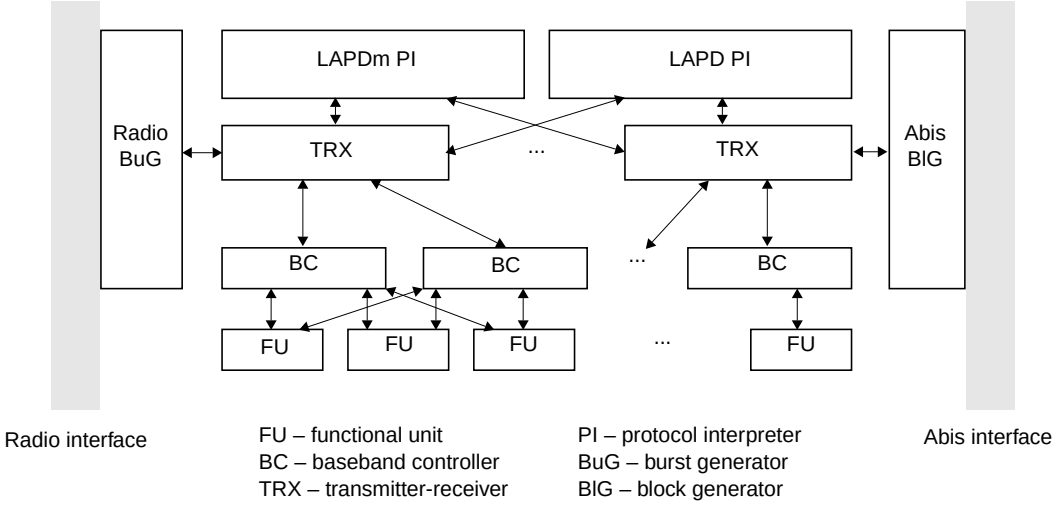

Figure 1. Structure of the GSM BTS model

Additionally, the model has two more packages, “Globals” and “Shell”. Thus, the model consists of eight packages:

- *Abis*,
- *BasebandCtrl*,
- *FunctionalUnits*,
- *Globals*,
- *PIs*,
- *Radio*,
- *Shell*,
- *TRX*.

2. Description of Packages

First, the role of the two additional packages is described. Next, the mechanics of the packages depicted in Figure 1 is explained starting from the interfaces towards the inner packages.

2.1 The *Shell* Package

The *Shell* package was introduced in order to interface the model with the environment (designer supplied events) easily. The package consists of a single class, *Shell*, which implements associations to various entities in the model. All the events to be sent to the model from the exterior are sent to the *Shell* object, which in turn dispatches them to the appropriate entities. The messages (events) which can be sent at the moment to the Shell and implicitly to the model are:

- *evAccess(AccessReason reason)*,

- *evGrant(int trx, int timeslot, ChannelMode mode),*
- *evDeallocate(int trx, int timeslot).*

2.2 The *Globals* Package

The *Globals* package consists of utility classes, global functions and variables used in the entire model. The classes are:

- *BitAddressableBuffer,*
- *Buffer,*
- *GlobalClock,*
- *Queue,*
- *BTSConfig.*

Several global functions provide a means to determine the position in the frame structure on the radio interface. Such functions are *getTSN(), getFrame()* etc. The global variables are the clock (*clk*) which gives the timing on the radio interface and the variable that holds the initial channel configuration of the BTS (*config*). Of particular interest is the *BTSConfig* class. The information of greater importance contained in the *config* object is the following:

- The configuration for each TRX, i.e. the channel types for each physical channel (*TRXconfig*),
- The number of carriers (*noOfFreq*),
- An indication to the TRX responsible of the beacon channel (*beacon*).

2.3 The *Radio* Package

The *Radio* package corresponds to the *Radio BuG* block in Figure 1. The main class in this package is *RadioSubsytem*. It registers itself to the GlobalClock, so it is notified every time a burst interval (577 μs) has passed. When it receives such a notification it commands to its *Demodulators* and *Modulators* to generate (emit respectively) a burst. The generated bursts are then fed to the *TRX*s by means of the 1:N directed association which exists between *RadioSubsystem* and the *TRX*s. This briefly described mechanism of simulating the BTS radio subassembly is depicted in the UML sequence diagram *DispatchingMechUpLk* (see model). Different types of modulators and demodulators exist in the model because different types of bursts are generated depending on the physical channel type. The specialization hierarchy of the modulators and demodulators is depicted in the *Modems* UML object model diagram. Besides the *RadioSubsytesm* and the modulators/demodulators classes, classes modelling the bursts were implemented. The *Burst*s objects aggregate several *BitAddressableBuffer* objects. Methods for retrieving the useful parts of the bursts were provided (*getInformation()*). The specialization hierarchy of the *Burst*s classes is depicted in the *Bursts* object model diagram (see model).

2.4 The *Abis* Package

The *Abis* package is similar to the *Radio* package. It corresponds to the *Abis BlG* in Figure 1. The package main class is *BlockGenerator.* It generates *AbisBlock*s according to the multiplexing scheme on the Abis interface. Such an *AbisBlock* object provides methods to extract different kind of blocks depending on the channel mode they are addressed to (*toF24Block(), toSpeechBlock* etc.). Those blocks (blocks for F96 channels, speech channels, and so on) are then dispatched to the *TRX*s by means of the 1:N directed association which exists between the *BlockGenerator* and the *TRX*s. The *evAbisBlock(AbisBlock *block, int physChSlot)* event carries the block together with an indication for its destination on the radio path. It is *BlockGenerator*'s responsibility to set the *physChSlot* parameter appropriately.

2.5 The *Functional Units* Package

The *FunctionalUnits* package gathers a large number of functional units from the baseband processing subassembly. The units and their inheritance relationships are depicted in the *Convolutional codecs*, *CRCs*, *Deinterleavers*, *Interleavers* and *ParityEncDec* object diagrams. Despite the large number of those units, they all exhibit a simple interface, which basically consists of two messages: one that starts the operation, the source of which is a baseband controller, and one that notifies the baseband controller about the completion of the operation. The processing algorithms are specified in C++ functions, with self-explanatory names (*interleave*, *computeCRC* etc.). The processing-completion event is emitted after a specified time interval, which simulates the

execution delay of the unit. The time interval can be specified for each unit individually at construction time and it is stored in an attribute in the superclass of all functional units, *ProcessingUnit*.

2.6 The *BasebandCtrl* Package

The *BasebandCtrl* package corresponds to the *BC* blocks in Figure 1. For every channel mode there exists such a baseband controller, which specializes one of the two more general controllers *UpLkCtrler* and *DownLkCtrl*. A controller groups and manages two or three functional units, depending on the channel type. The structure of a generic baseband controller is shown in the *Generic Controllers* object diagram (see model), which we consider quite important for the general understanding of the baseband internals. The activity of a downlink controller is triggered by an *evBlock* event. This event is sent by the TRX or, more precisely, by a *physical channel* managed by a TRX. The uplink controller has a similar behaviour. When the controller receives an *evBursts* event, it will command the appropriate functional unit. Data communication is modelled by means of event parameters. An *evBursts* event carries four bursts. It is the responsibility of some TRX managed entities in the TRX package to buffer those four bursts. However some deinterleaving algorithms need 8 or even 22 bursts in order to build up a block. It is then the deinterleaver responsibility to further buffer groups of four bursts in order to reach the needed number of bursts to fire a deinterleaving operation. The consequence is that the uplink controller behaviour will slightly differ from the downlink one. Thus, not every operation-starting event (*evBursts*) will actually fire the entire processing chain. The *evTransient* event was introduced in the uplink controller statechart in order to capture this aspect.

Further specialization of generic controllers, according to specific channel modes, is depicted in the *Controllers* object diagram (see model).

2.7 The *TRX* Package

Although it is not the highest in the control hierarchy, we consider the *TRX* package as being the most important unit in the model. The protocol interpreters are viewed more as servers for the TRX requests.

The *TRX* package consists of several *PhysicalChannels* and the *TRX* class. A *TRX* manages eight physical channels. It has to keep track of the channel *types* and states (allocated or not). By channel *type* we mean *combined common*, *common*, *full traffic*, *half traffic*, and *signalling*, as opposite to the channel *modes* which characterize the data semantics of the *logical* channels carried by the physical ones. The channel modes can be *speech*, *F96*, *F48*, etc.

The TRX exhibits a rather complex behaviour with three orthogonal components:

- signalling (channel allocation and deallocation),
- dispatching of uplink bursts,
- dispatching of downlink blocks.

The *evAllocate* and *evDeallocate* events from the signalling component are sent by an entity in the *PIs* package, which will be described in the next section.

The behaviour of the downlink component is the following: when the TRX receives an *evAbisBlock(AbisBlock *block, int physChSlot)* event, the TRX extracts the useful information, and subsequently it will forward this useful data block to the *PhysicalChannel* on the *physChSlot* timeslot. Extraction of the useful information is done by means of the methods provided in the *AbisBlock* class (see 2.4) and according to the channel type and channel mode information which are stored in the *TRX* class and *PhysicalChannel* subclasses respectively (see the TRX *extract* method). Forwarding is next done by means of the *evABlock* event sent to the appropriate *PhysicalChannel* class.

The activity on the uplink component is triggered by an *evBurstTick* event. This event is sent by the *RadioSubsytem*, which in turn receives it from the *clk* singleton object (of class *GlobalClock*). This event only specifies that an interval equal to a burst period has elapsed and that the *RadioSubsystem* might have generated a burst. It is the responsibility of the TRX to determine the “position” in the frame hierarchy on the radio path, in order to dispatch the burst correctly. This is done by means of the global functions *getTSN()*, *getFrame()*, etc., in the *Globals* package (see 2.2). If the TRX is the one in charge of the beacon carrier it will interpret differently the bursts on timeslot 0 and will enter the *access* state. If an access is requested, it will send a request message to an entity in the *PIs* package. This will be discussed in the following paragraph. The access request is indicated by the *accessAttempt* boolean variable which is set by the shell when it receives an *evAccess* event and unset by the TRX when it takes the request into consideration. Otherwise, if the conditions for access request are not met, the TRX checks whether a channel is allocated on the current timeslot. If not, the *evBurstTick* is ignored. Otherwise, the TRX gets a burst from the *RadioSubsystem* and dispatches it to the *PhysicalChannel* on the current timeslot.

The entire model is set up according to the information stored in the *config* singleton object of class *BTSConfig* (see 2.2). The TRX configuration, stored in the *config* object, is sent as a parameter to the TRX constructor. According to this configuration, the eight *PhysicalChannels* managed by the TRX are constructed. There are five such *PhysicalChannels*:

- *CombCommonCh* (combined common channel),
- *CommonCh*,
- *FullTrafficCh*,
- *HalfTrafficCh*,
- *TACH8Ch* or SDCCH (dedicated signalling channel).

The *CommonCh* physical channel carries the RACH, FCH, SCH, BCCH, PAGCH logical channels. The *FullTrafficCh* carries the TCH/FS (full speech), TCH/F9.6, TCH/F4.8, TCH/F2.4 logical channels. The *HalfTrafficCh* carries two TCH/H4.8 logical channels or two TCH/H2.4 logical channels or one TCH/H2.4 and one TCH/H4.8 logical channel. A *TACH8Ch* carries eight SDCCH logical channels. A *CombCommonCh* carries the RACH, SCH, FCH, BCCH, PAGCH and four SDCCH logical channels. The configuration coded in the model uses only *CommonCh*, *FullTrafficCh* and *TACH8Ch* physical channels, but the other ones should work also.

A *physical channel* relates to *baseband controllers* (see 2.6) for the logical channels it carries. Additionally, a physical channel class contains *buffers* (2.2) where it buffers the bursts to be deinterleaved. Remember that a deinterleaver works on groups of four bursts. Dispatching of incoming bursts to the appropriate logical channel, i.e. baseband controller, is done by means of interrogating the “position” in the frame hierarchy (see the *dispatchToLogCh* method).

The partitioning of information between *PhysicalChannel*s and *TRX*s is made as follows. The TRX stores the *types* of the eight physical channels it manages. It also keeps track whether a physical channel is allocated or not. A physical channel is allocated when it carries at least one logical channel. Additionally, a TRX keeps track of the subtimeslot, in case of half traffic channels (the *round* member variable). The *PhysicalChannel* stores only the channel *mode(s)* of the logical channel(s) it carries and their status (allocated or not). See the attributes part of the classes and the corresponding accessor and mutator methods.

At allocation the needed resources (baseband controllers and the appropriate functional units) are dynamically created. Thus, these units can be seen as *virtual* units, and not necessary as physically existing ones. In reality, two or more such virtual units can be mapped on existing physical units, if the time multiplexing scheme allows their simultaneous usage.

The split between *TRX* and *PhysicalChannel* was made because of complexity management reasons and not because of functionality reasons. We consider that they fulfil the same functions. This is why we did not depict *PhysicalChannel* as a separate block in Figure 1.

2.8 The *PIs* Package

This package contains the classes needed for modelling the LAPDm and LAPD protocol interpreters. The *LinkSender* and *LinkReceiver* classes play key roles in the mechanics of signalling transmission. They model the acknowledged transmission mode, the sliding window concept as well as frame fragmentation and reassembling. Both classes inherit from the *Link* class, whose only role is to concentrate all the common data and utility methods of the two subclasses. Thus, both classes use the following attributes:

- *frameLength*, (Length of one frame, 184 bits for LAPDm and 260 bytes for LAPD),
- *repetition*, (A frame number can take values in the interval 0–*repetition*-1),
- *windowSize*, (Maximum number of frames which can be transmitted before an acknowledge is received),
- *segments*, (Buffers for storing the frames which result from a upper layer message decomposition),
- *segs*, (Number of frames, actually the *segments* array length),
- *transmissionBuffers*, (An array of *windowSize* buffers which store the yet not acknowledged frames),
- *transmissionTime*, (The time needed for sequencing the frame bits on the line).

The *LinkSender* class contains the following attributes:

- *timeout*, (How much the sender must wait for an acknowledge before signalling an error and trying a retransmission),
- *sent*, (How many frames were already sent, not necessarily already acknowledged),

- *va*, (*va*-1 equals the last acknowledged frame),
- *vs*, (*vs*-1 equals the last frame sent),
- *timerRunning*, (Boolean variable which indicates whether the timer was already started or not. The timer is started whenever a frame is sent and the timer is not already running).

The activity of a *LinkSender* object is triggered by an *evSendMessage(LinkMessage *inf)* event. The message to be transmitted is fragmented in frames which are stored in the *segments* buffers. *windowSize* frames are next downloaded in the *transmissionBuffers*. The sender starts then transmitting the frames. Whenever a frame is sent and the timer is not already running, it will be started. Transmission stops when *windowSize* frames were sent without being yet acknowledged. In such a situation the sender enters the *cannotTransmit* state. Two events can cause a transition from this state: either an *evError* or an *evAck* event is received. The *evError* message is received when the timer indicates a *timeout* elapsing. In this case the sender sets *vs* to *va* and starts retransmission. When an *evAck* is received, the timer is reset, all acknowledged frames in the *transmissionBuffers* are discarded, new ones are downloaded from the *segments* buffers and *va* is updated. If there are still unacknowledged frames the timer is restarted. The sender returns to its initial state (the one which is exited only when it receives the *evSendMessage* event) when all the frames were sent and acknowledged. Then all the buffers are purged.

The *LinkReceiver* class contains the following attributes:

- *entireMessage*, (Boolean variable which indicates whether an entire message was received or not),
- *inf*, (Buffer where the received upper layer message is assembled),
- *toAck*, (Integer variable that holds the number of next frame to acknowledge),
- *vr*, (vr-1 equals the last received frame),
- *received*, (How many frames were already received. Corresponds to the *sent* member variable in the *LinkSender* class).

The *LinkReceiver* is composed also of a *Queue* which stores the acknowledge messages. This queue is used in order not to lose acknowledge commands. Such situation could be possible due to the fact that transmission of acknowledges takes *transmissionTime* milliseconds.

The activity of a *LinkReceiver* is triggered by an *evGetFrame(LinkMessage *message)* event. Regardless of the frame sequence number, the frame is acknowledged. If the frame is also the one the receiver waits for (*message->NS == vr*) then the frame is stored in the *segments* buffers and *vr* is updated. If the last frame from a message is received then the receiver returns to its initial state, a *LinkMessage* is assembled from the *segments*, the algorithms variables reinitialized, the *segments* purged and the upper layer notified.

The *LinkSender* and *LinkReceiver* classes relate to the *L2Channel* class. The latter models the transmission channel. Such a channel is characterized by the following attributes:

- *recDelay*, (Channel latency in carrying a frame in a particular direction),
- *sndDelay*, (Channel latency in carrying a frame in the opposite direction),
- *recEff*, (Receiving efficiency, i.e. the percent of transmitted frames in the first direction),
- *sndEff*, (Sending efficiency, i.e. the percent of transmitted frames in the opposite direction).

A *L2ChannelHelper* is created whenever a frame is conveyed by the channel. Actually the helper is the one that performs the transmission. Having one helper for every frame transmission, transmission delays can be specified on a per frame basis. The consequence is that frames can arrive at their destination in a different order than they were sent. Thus, a feature of packet data networks can be modelled. However this feature is not used in the model.

By parameterizing *LinkSender* and *LinkReceiver* by means of inheritance, senders and receivers for the particular LAPD and LAPDm protocols are created (e.g. *windowSize* equals 8 for LAPD and 1 for LAPDm, *repetition* equals 8 for LAPDm and is negociable for LAPD — defaults to 128 in our model).

For a more synthetic description of those classes see the *Linkers* object diagram. The *communication* sequence diagram depicts the sequence of events which are exchanged between the main entities in order to convey a layer 2 message.

The information carried by these layer 2 entities is encapsulated in an *LinkMessage* object. A *LinkMessage* consists of a *LinkAddrField*, a *LinkCtrlField* and a *BitAddressableBuffer* which models the information field. The former two components provide methods for extraction of various information (like, for example, sequence number). By specializing and extending the *LinkMessage* class the *LAPDMessage* and *LAPDmMessage* classes are built. For a more detailed view of the messages composition and inheritance relationships see the *Messages* object diagram in the model.

A *LinkLayer* class groups a *LinkSender, LinkReceiver* and a *L2Channel* class. The specialized classes *LAPDLkLayer* and *LAPDmLkLayer* result from *LinkLayer* by means of inheritance. They are also depicted in the *Linkers* object diagram.

On top of those layer 2 entities the *RRLkLayer* class is built. It models the *radio resource* link. This layer 3 entity implements an association to a *LinkLayer* object (actually to one of *LinkLayer* subclasses). The *RRLkLayer* interface consists of two events:

- *evSendL3Message(RRMessage *message),*
- *evGotMessage(BitAddressableBuffer *message).*

The former event is sent by the *RRLkLayer* exterior (is a command), while the latter is sent by the lower layer *LinkLayer* which actually only forwards the same event received from a *LinkReceiver* (is a notification). When *RRLkLayer* receives an *evGotMessage* it calls the *interpret* method to interpret the contents of the *message* according to the radio resource protocol. This is simply modelled by associating to the radio resource protocol procedures numerical values which in turn are associated to the *BitAddressableBuffer.*

There exist two specialization of the *RRLkLayer* class: *RRLkLayerFromBSC* and *RRLkLayerToBSC*. Actually, they only add some relations to other entities in the model creating thus layer 3 links between them. *RRLkLayerFromBSC* sends layer 3 messages from the *Shell* (which plays the role of the environment, implicitly of the BSC) to *config->noOfFreq TRX*s. The *RRLkLayerToBSC* conveys messages from the *TRX* responsible of the beacon carrier to the *Shell*.

3. A Possible Scenario

An *evAccess* event is sent to the *Shell* (which plays the role of a mobile station is this case).The *Shell* will set the *accessAttempt* boolean variable of the *TRX* responsible of the beacon carrier. Upon access attempt detection, the *TRX* will request the transmission of a layer 3 message from its *RRLkLayerToBSC*. When the message arrives, the *Shell* (which plays the role of the BSC now) is notified. Next, the user will send an *evGrant(int trx, int timeslot, ChannelMode mode)* to the *Shell.* The *Shell* will request the transmission of an allocation layer 3 message from *RRLkLayerFromBSC*. When the message arrives, the *RRLkLayerFromBSC* notifies the *TRX* specified in the message. Next, the *TRX* allocates the channel, and updates its member variable values (*allocated* etc.). Next, whenever a burst or block arrives for the newly allocated channel timeslot, the burst or block is finally dispatched to the *functional units*, after being “guided” and perhaps buffered by the *physical channels* and *baseband controllers.* This whole scenario is depicted in the *Channel Activation Protocol* sequence diagram. An intuitive GUI (*BTSExec/bts.exe*) was built in order to support this scenario.