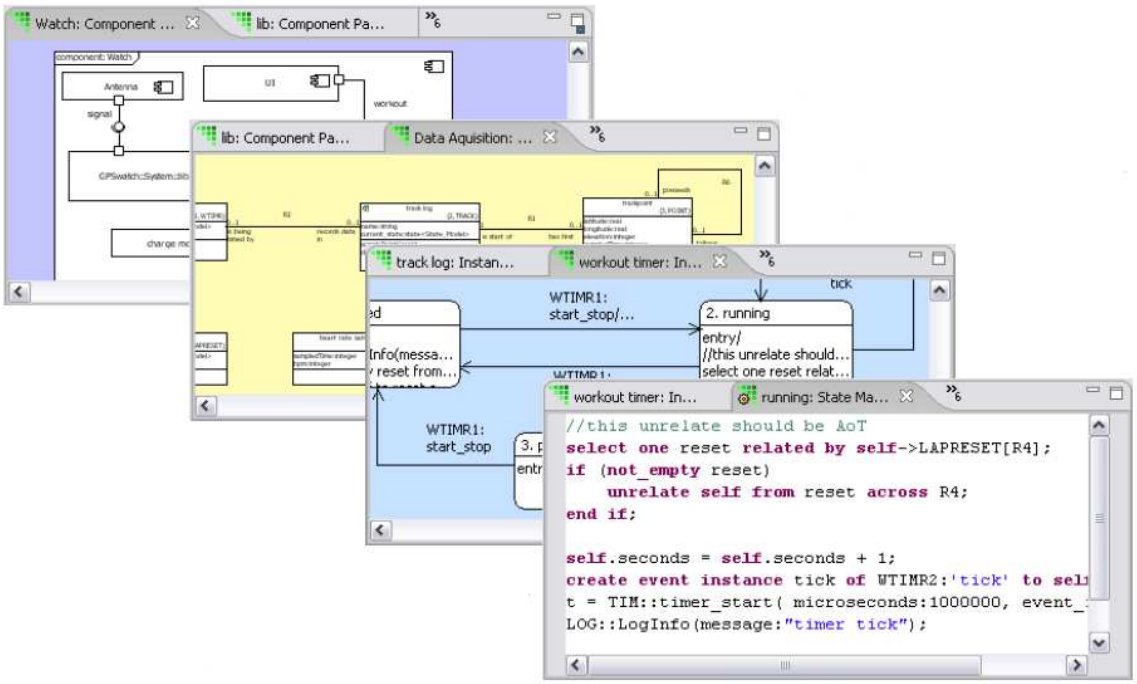
CHALMERS | UNIVERSITY OF GOTHENBURG



Executable System Design

Master of Science Thesis in software engineering and technology/management

Amir M. Najafzadeh
Shohreh Farahani

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2011

Executable System Design

Amir M. Najafzadeh
Shohreh Farahani

Examiner: Gerardo Schneider
Supervisor: Rogardt Heldal

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The cover picture has been taken from Mentor graphics and illustrates the hierarchy of different diagrams within an executable UML model.

Department of Computer Science and Engineering
Göteborg, Sweden June 2011

Abstract

Software models are important in building large software systems. Even though these models are used to simplify the software system, they can be in themselves, quite complicated. It is not all clear how to build these software models in the best way. This work tackles that problem. We show how one can start with an abstract platform independent model (PIM) and transform it into the complete PIM. The complete PIM is the one which can be translated into a platform specific model (PSM). At each level of the abstraction the PIM is executable and testable. We have done a case study within Ericsson.

Acknowledgments

This research project would not have been possible without the support of many people. We wish to express our gratitude to our Chalmers supervisor, Dr. Rogardt Heldal, who has been abundantly helpful and offered invaluable assistance, support and guidance.

Deepest gratitude is also due to our supervisors at Ericsson's Baseband Research group, Mr. Martin Lundqvist and Mr. Roland Carlsson, without whose knowledge and assistance this study would not have been successful.

We would like to thank the Baseband Research group at Ericsson AB for welcoming us to their inspiring work environment.

We would also like to convey thanks to PhD student Mr. Hakan Burden for his kind help and comments with the documentation.

Table of Contents

1. Introduction

Moving from more concrete and machine level languages like assembly and C up to object oriented languages like C++ and JAVA was a significant step to reduce the complexity of systems. Now a new era has risen and complex systems can be designed and analyzed even on higher abstraction levels - the era of software models.

Using platform independent models [7] (PIM) has become a way of handling the complexity of building software systems. This is due to the fact that we can abstract away from platform details. PIM is an abstraction level within the Model-Driven architecture (MDA) [5] [7]. It can be transformed into platform specific models [7] (PSM). This provides a clear separation of concern between PIM and PSM, since a PIM does not contain any platform details which are required by the PSM to be executed on the particular platform.

Even though a PIM is an abstraction from platform details, it can be hard to produce for a complex system. Part of the solution can be to use executable UML [15] that permits building executable PIMs. This makes it possible to validate the PIM. However this is not enough, we also need a methodology to build these executable PIMs.

Our belief is that a complete PIM contains too much detail to be produced in one go. There are many ways of building a PIM, for instance we can build it iteratively using well known agile methods [17] [21]. In this report we present a methodology defined within Ericsson and Chalmers which can be used to obtain a complete PIM.

The key concept is to split the PIM itself into several abstraction levels. Thus we start with an abstract PIM which will be refined in several iterations to produce the complete PIM. The complete PIM is the one which can be translated into a PSM.

1.1. Motivation

Today, complex software systems are emerging in various domains of everyday life. This is why software designers are looking for different techniques to handle this complexity. It seems quite feasible to create platform independent model for a large software system before jumping into the common process of the development.

As mentioned before the complexity of the software systems results in the complexity of the PIM. It is therefore quite hard to create a complete PIM from scratch. This means that a sufficient abstraction is needed to produce a complete PIM.

Difficulties to create a complete PIM in various IT-related companies have been a motivation for this research.

1.2. Aim

The aim of this research is to investigate a new methodology to produce a complete and fully tested PIM for a large and complex system. The main goal is to find a way to make systems less complex by abstracting their key issues and then breaking them

down in an iterative manner. The methodology which is introduced and validated in this research goes from a complex problem to an abstract solution and then adds detailed complexities step by step during each iteration until a complete PIM has been reached.

Note that the main divergence between this methodology and the common ways for designing a platform independent model is that the common ways of producing models are valuable when the system is small or simple, but in cases where you have a big and complex system, creating a complete PIM would face difficulties.

Moreover we believe that by applying this methodology several PIMs can be produced for different entities in a system not only will the produced PIMs be fully tested and executable but they will else be integrated within the entire scope of the system at the end of each iteration. Thus the risk of error propagation caused by late integration of newly added features should be reduced. The latter has not been validated by this research because we have only focused on one component of a system.

Furthermore, the introduced methodology is an agile-like process, because it follows the same principles of agile processes and agrees on the agile manifesto. For more information see section 2.1.

1.3. Contribution

The idea for the methodology explained in this thesis initially came from Dr. Rogardt Heldal and the Ericsson's researchers Martin Lundqvist and Roland Carlsson. They were our supervisors and guided us through the entire research. We, Shohreh Farahani and Amir M. Najafzadeh, validated the methodology in a case study conducted at Ericsson and created this paper with their assistance. The methodology is explained in theory within section *3. Executable system design* and it is also validated and applied practically in section *5. Case Study*.

1.4. Research Approach

The research started by defining the goals and motivation for the project. During the first meetings we discussed the problem that we wanted to solve. The idea and potential problems which may arise during the development of complex systems that exist today were identified and an initial idea for a methodology to solve them was defined.

After defining the method we decided to validate the research by fulfilling a case study. The case study chosen by Ericsson's supervisors was the radio link control protocol which resembled a complex system. This case study was chosen because it is a complex system that has been already implemented and has a complete specification.

After defining the case study, the research phase of the thesis began. The research focused on learning the specification. We spent the first weeks of our research to study the BridgePoint [8] tool. During the first phase of the research, we read various papers and books about model driven development, UML, executable UML, agile processes, QuickCheck testing etc.

The next step was to decide how to start the implementation and modeling. Moreover, the various ways to test the system properly was discussed on the meetings. Thus, the research extended in two different dimensions, the test framework and the implementation part. We decided to follow a test driven development [4] and use the BridgePoint tool as a white box testing environment and then at the end of the iterations perform an automatic black box test.

The process of producing the PIM followed after brainstorming about the problem domain. The starting point was to design mind maps of, and to discuss the main architecture of the case study. After defining the number of components and their interfaces we created a class diagram.

Each iteration of our process ended with an automatic test of the produced PIM. At the end of each iteration, we presented our PIM to our supervisors and discussed the next iteration activities. Generally we had regular one hour meetings at the end of each iteration.

Our process of implementation was similar to that of a pilot and a copilot [13]. On each occasion one of us was responsible for drawing our models on the whiteboard and then we discussed our solution. At the point of agreement, the other person was responsible for designing and programming the solution into the computer. These responsibilities were swapped each day.

The first month of the project we tried to tackle the problem using the common agile ways for creating applications. Doing so made us confident that this way of designing a complete PIM for a complex and large system has some draw backs. The produced PIM was not a reliable product (for more information see section *7.Reflection*). Therefore, we changed our methodology, which is discussed in section *3.Methodology*.

In addition to our meetings with the supervisors, we attended an online conference which is organized by MentorGraphics. The conference was mainly about the power of the model driven development. The discussions and ideas during the meetings were gathered by us and used in the documentation part. We started to create the report from the third month of the research. The documentation phase was an iterative process that improved section by section with the help of our supervisors.

2. Background

In this section, two important techniques that has been used in our research is described. First, the summary of the well-known manifesto for agile software development. Then, a brief explanation of Executable UML as a modeling language for platform independent system development.

2.1. Agile Manifesto

Agile [17] [21] approaches provide higher flexibility in the development process, with the aim to make the communication between developers and customers easier in order

to be able to change the system according to stakeholders' feedback. The changes should be applied and verified in each iteration. These small and rapid iterations assure the quality of the system requirements.

All agile processes follow a set of principles and agree on the agile manifesto. As Kent Beck et al explained the agile manifesto is:

- **"Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan"

That is, while there is value in the items on the right, we value the items on the left more. Our methodology is an agile process because as mentioned in the manifesto, we focus on the working and executable PIM which also has the ability to be used as a complete specification rather than dedicating too much time on the documentation. Also we have regular meetings with customer representative and other stakeholders throughout the design process. Finally, by focusing on the PIM and testing the executable product in an early phase and abstracting complex requirements it is easier for us to take the proper action when a change is needed.

2.2. Executable UML Models

According to [15], Executable UML (i.e. xtUML) is one of the major innovations in software development and can be defined as a foundation for the Model-driven architecture. Its main goal is to create a comprehensive model of the solution which is independent from the implementation platform.

UML [1] [16] is not executable and although it is a useful modeling language, it cannot close the gap between specification and implementation. Therefore, xtUML evolved to eliminate this gap by adding executable semantics to the UML's graphical notation. By doing so, it made the specifications runnable. Additionally, a model can be tested at a high abstraction level and code can be generated from the tested model.

An xtUML specification consists of various diagrams that define the real world under study as models. The fundamental elements in executable UML are defined as follows:

Component Package Diagram: The component diagram is the highest abstraction within the system that can specify the main architecture of the system. This diagram mainly illustrates different subsystems in the system and how they connect to each other via different interfaces to form the whole system. Notice that the only way for communication between different components is by sending signals or messages through these well-defined interfaces. *Figure 2.2.1* depicts a component package diagram that consists of two components, named *RLC* and *TestFramework*. These components are connected together through interfaces *PDCPInterface* and *MACInterface*.

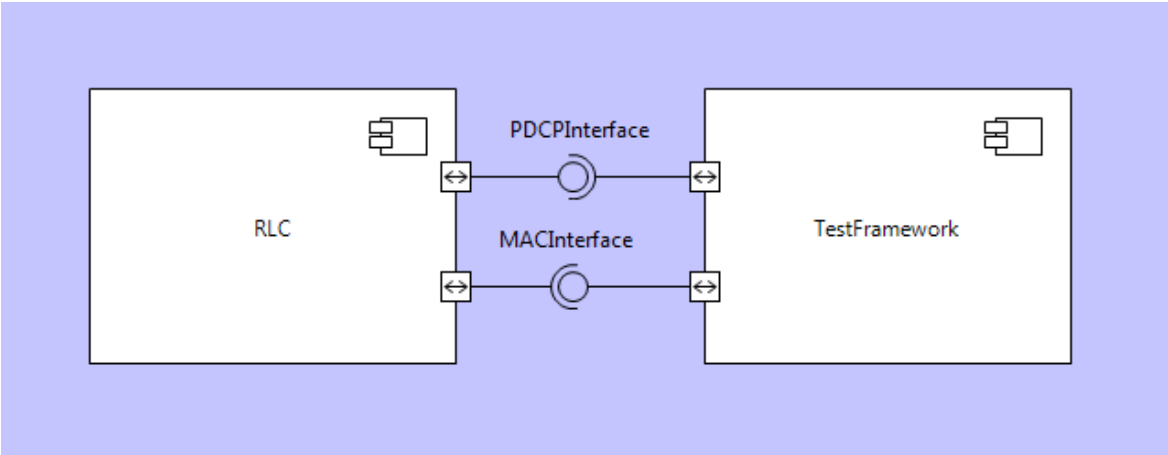


Figure 2.2.1 - Component Diagram

Class Diagram: Each component of the system consists of sets of conceptual entities that belong to that component. An abstraction of a set of entities that shares common characteristics and behaviors is called a class. A single characteristic of an entity is an attribute of the class and each behavior of the class is defined as an operation. The main difference in an xtUML class diagram is the class key letter which used by action language. Each class may also have a state machine. A class diagram contains various classes which relate to each other through various relationships. *Figure 2.2.2* shows a class diagram with three classes *RLC_TX*, *RLC_RX* and *RLC_Dispatcher*. Notice that *RLC_TX* and *RLC_RX* classes are related to each other with relationship *R1*.

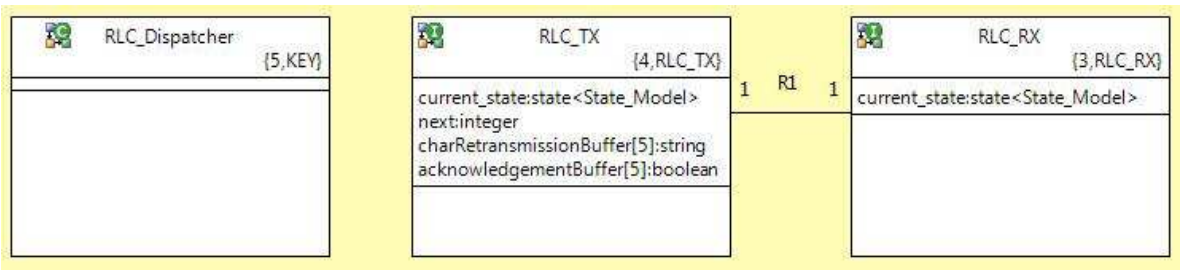


Figure 2.2.2 - Class Diagram

State Machine: A class might have a state machine. The state machine represents the life cycle for instances of the class. A life-cycle contains different stages that an instance of a class passes through over time. Therefore the instance may have different behaviors at various stages. State machines are categorized into *instance based* and *class based* state machines (the icon containing *C* letter in *Figure 2.2.2* shows class based state machine and the icon with *I* letter shows instance based state machine).

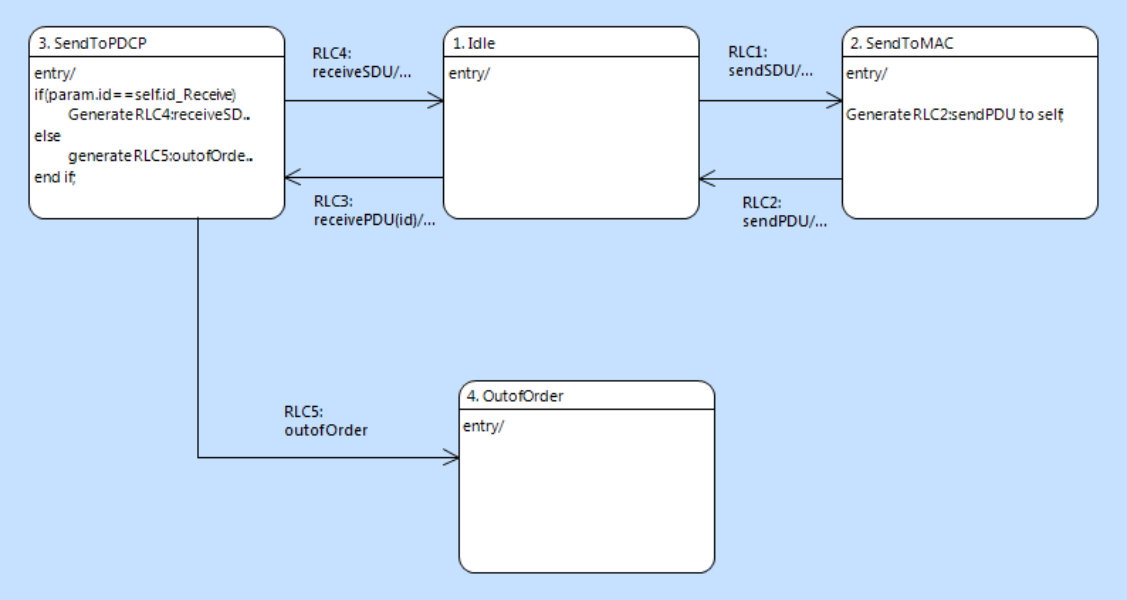


Figure 2.2.3 - State Machine

The major difference between them is that in instance based state machines there must be an instance of the class created to use the state machine, whereas class based state machines does not require any object, in other words the state machine will be created as soon as the whole system is created. Furthermore, instance based state machines per definition points out their own instance's attributes and operations.

The main constituents of a state machine are [15]:

- State: Representative of an object's stage.
- Event: Representative of a cause or an incident that forces the stage change.
- Transition: Specify the stage change of an object that forced by an event.
- Procedure: Operations or activities that are performed when an object enters or exits a stage.

Different states and transitions among them are shown in *Figure 2.2.3*.

Action Language (Object Action Language): Action language is a textual language that enhances graphical models. In order to model the dynamic view of instances, action semantics must be attached to the models. Technically, this is the action language that makes the models runnable by performing on attributes, invoking operations, raising events and so on. The action language is visible as written text on different states in *Figure 2.2.3.*

3. Executable System Design

Here, the new methodology to produce a complete PIM is introduced, but first let us take a look at the common iterative ways for the software design.

Now, the development of software systems is usually done in the way represented in *Figure 3.1 (vertically)*. Consider this example as a set of iterations that takes to build a

complete PIM for a complex system. As the figure depicts, each iteration starts with some functionality and the complete feature will be delivered at the end of iteration. Although the feature has been delivered however it has gone through an extreme difficult phase with too much details that could be abstracted away. Moreover, this feature has been poorly tested due to the amount of details it includes. The point is that one cannot be sure that the whole thing will not collapse in presence of other features.

In order to clarify the problem let us compare a complex software system to a human body. Human body is constructed by different organs. Each organ is constructed by cells and further down a cell constitutes of molecules, atoms, and at the end we can say an organ composed of billions of particles. Now, consider that we want to build a complex system just like a human body. What if we want to add a feature to this body like a simple walking?

Then it is necessary to build legs for the body and in the mentioned way to build that feature we have to consider about many details and there is no guarantee that the resulted feature would behave as we wanted. Back to our example, did we test all the details about elementary particles? In more general case assume that everything has been built and tested in a perfect manner, what if the whole system at the end (i.e. Body) rejects that entity (i.e. legs)?

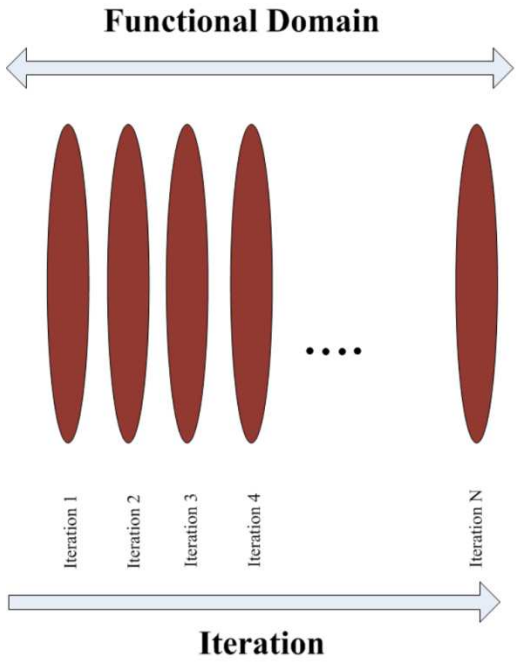


Figure 3.1 – The current iterative way for the software development

Now that you have grasped the problem let us take a look at our solution. As shown in *Figure 3.2* horizontal bars show the iterations. Going horizontally means that on each iteration a certain amount of abstraction must be considered. By abstracting away details, it is much easier to provide a fully tested product at the end of iterations.

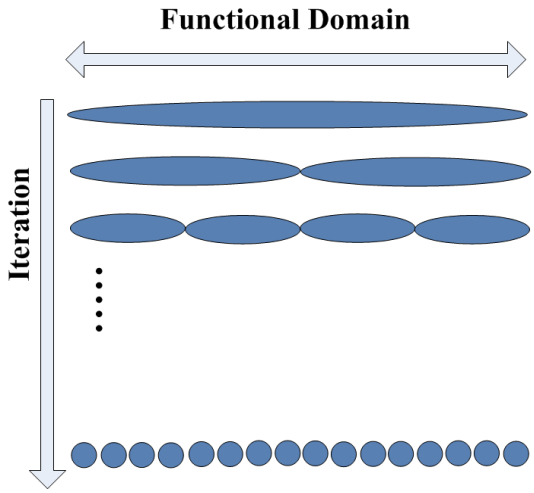


Figure 3.2 – The new methodology
(Segmented bars means that different task may be done by different teams)

In order to compare this to the human body example, let us take a look at what we need in order to add a particular feature. Let us consider the “walking” feature one more time. In the new way of producing the PIM, we can abstract details about different interacting muscles down to atoms and elementary particles. A good abstraction here might be the skeleton of the body that can walk. The system can be fully tested and guarantee the behavior that we wanted at the end of the iteration. Other iterations will use the current abstract level and add other details to the latest product. For example the next iteration can be dedicated to muscular framework of the body and be tested at the end.

The crucial point of this methodology is the correct way of selecting the level of the abstraction. It is important to know that the PIM itself is an abstraction of the underlying technology and with this method we want to bring abstraction within the PIM. This can be interpreted as an abstraction within an abstraction. Notice in our example we do not want to think about the color of the leg’s skin before considering bones for the leg.

We believe that this way of producing PIM is easier than common ways of producing PIM. By applying this methodology, not only a complete PIM will be produced, but at the end of each iteration we can fully test the system and gain a confidence that the PIM is executing in the way that we have expected. In plain words, the main divergence between the common approach and the new methodology is that the first one focuses more on the feature increase while the second one focuses on the architectural view of the system.

Along with simplifying the complex solution via the abstraction of functional requirements, this methodology reduces the cost of the development process and improves the quality of the final product. This is because an executable PIM which has been tested step by step during the design process will result in reducing the error propagation and further unexpected behaviors. As a conclusion, the final product has a

higher quality in terms of reliability and on the top of that because the PIM is independent from the underlying platform it has the portability feature.

The recommended testing for this methodology is a black box testing of different components which has a “generate and validate” pattern and it is similar to the QuickCheck approach [23]. The test framework is explained in detail in section *5.3 Execution.*

4. Research questions

The goal of this research is to investigate a methodology for designing platform independent models in support of large and complex software systems. The following question is answered after the case study has been completed (see section *6. Results*):

- How should we model a complex system in order to reach a complete and executable PIM?

5. Case study

This section validates the research’s goal by applying the suggested methodology on a complex software system in industry. The following subsections explain how the validation part initiated. Also the domain of the case study is briefly explained and finally the main body of the development process is put in plain words. In section 6, the results of the research are presented which are validated by the case study.

5.1. Method

The findings that are presented in this research were reached by following the empirical research method using the case study analysis strategy. As mentioned, the process of modeling follows the agile manifesto and it started by brainstorming over the domain specification.

The main body of the design and analysis was done on the whiteboard by using a graph notation which is very convenient to analyze state machines. Similar to agile processes we had regular meetings with stakeholders throughout the iterations where the analysis and new ideas discussed amongst stakeholders. From there on, the analyzed results were implemented via BridgePoint, a tool for creating executable models.

In addition, we have followed a test-driven development [11] therefore; every iteration was started by designing a test case before modeling various features.

5.2. The RLC Domain

Radio link control (RLC) is a protocol layer in telecommunication systems. The main objective of this layer is the flow control and error recovery. RLC works in three different modes depending on how data needs to be transferred. TM (Transparent mode) is more suitable for carrying voice because there is no header attached to the

data and data delivery is not guaranteed. UM (Unacknowledged mode) is suitable for streaming traffic because data can be segmented and a header will be added to the data, but there is still no delivery guarantee. AM (Acknowledged mode) is the reliable data transfer, because in addition to segmentation and headers this mode contains the sequence delivery service. AM mode therefore is more suitable for TCP traffic. There are two neighboring layers that RLC communicates with them. The upper layer called PDCP, which sends/receives SDU packets to/from RLC. The lower layer called MAC and it is responsible for delivering PDU packets created by RLC to its peer. Bear in mind that MAC sends a packet using the physical layer and packets cannot be sent or received by RLC unless MAC layer grants a transmission opportunity. For more information about RLC see [22].

5.3. Execution

The RLC system was implemented in nine iterations, and focused on the AM mode of RLC which is more complex and challenging than other modes. Due to the lack of time some functionalities of the AM RLC, such as segmentation and concatenation of packets has not been implemented. However, the implementation of RLC was not the main goal of the thesis and the remaining parts could be accomplished using the same methodology in a few iterations.

The following subsections describe the implementation process along with some notes and recommendations. Bear in mind that for the sake of simplicity many attributes, variables, flags and algorithms are omitted in the explanation. For more information see Appendix A which contains codes for the final implemented system.

5.3.1. Iteration 1

Duration: 4 Days

Goal: Define the structure of the test framework, RLC and interfaces among them

The first iteration started by defining the basic structure of the system on component level. As shown in figure 5.3.1.1, RLC defined as a separated component and both the upper layer PDCP and the lower layer MAC are embodied within a *TestFramework*.

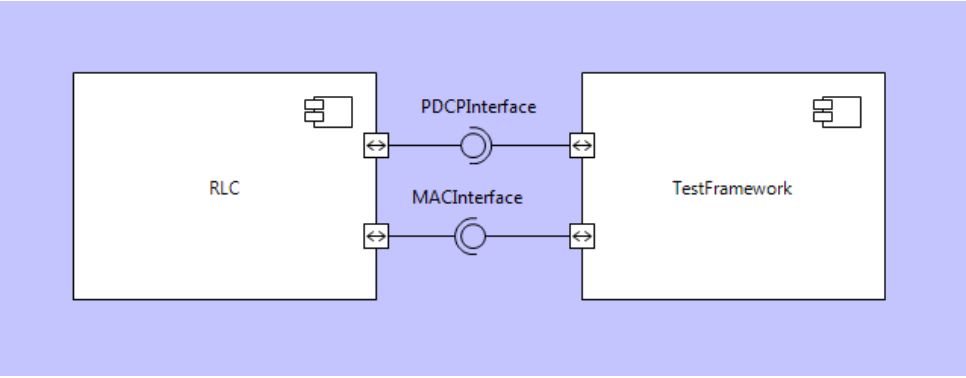


Figure 5.3.1.1 - Component package diagram

Bear in mind that each of the mentioned layers can be accessed only through their specific interfaces. Therefore two different interfaces with set of various signals are defined for each layer (Figure 5.3.1.2).

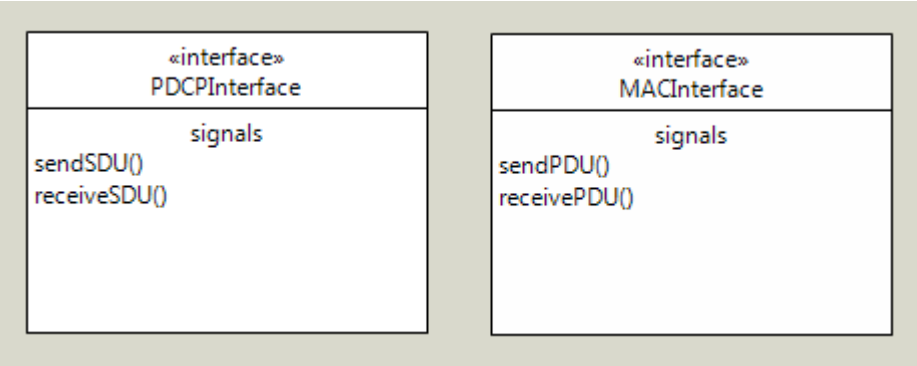


Figure 5.3.1.2 - Interfaces between RLC and TestFramework

Within the *TestFramework* a class diagram is defined with three different classes.

- Verifier: This class is the main test case of the system. In the upcoming iterations, the name of the class is changed.
- Tracker: The main functionality of this class is to keep track of test case operations (e.g. number of runs, total run, and later to set up the entire system, etc).
- Random: This class is used as a random generator which randomly generates one of the outgoing signals from the test case.

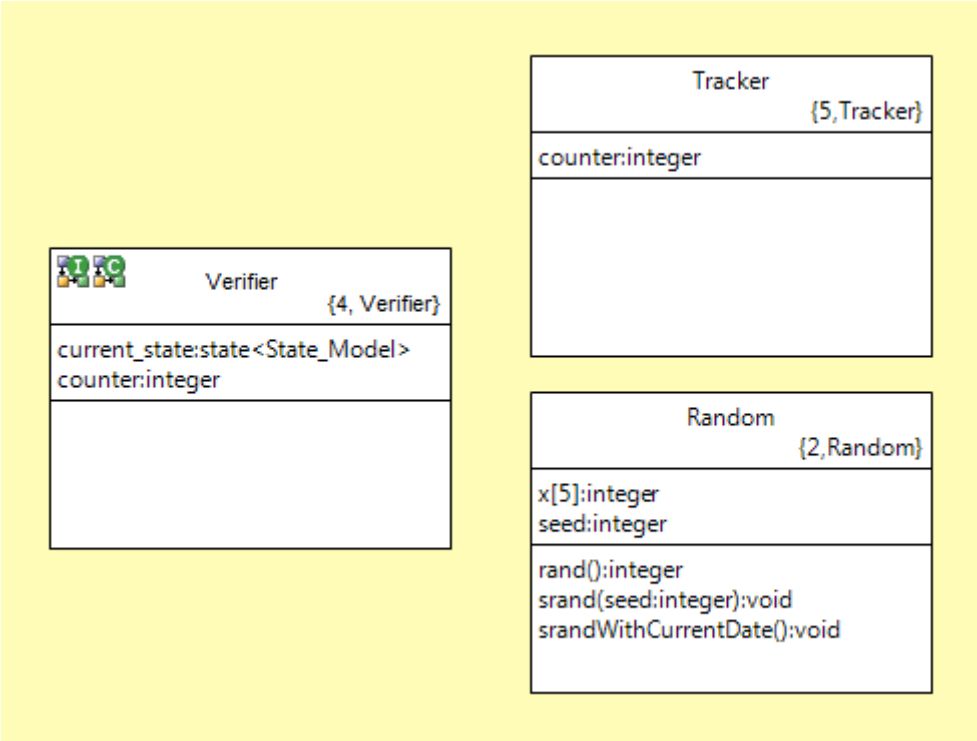


Figure 5.3.1.3 - TestCases class diagram

Among the mentioned classes, *Verifier* has two state machines (notice the icons for different state machine on class *Verifier* shown in figure 5.3.1.3; the class based state machine is represented by an icon with letter *C* on it). The class based state machine is responsible for catching signals from the other component (i.e. RLC) and binds them to the relevant events (figure 5.3.1.4).

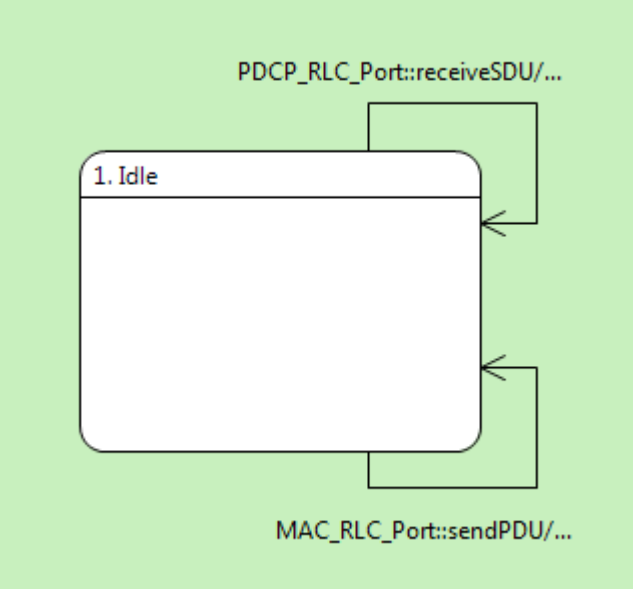


Figure 5.3.1.4 - Verifier class based state machine

Note: Creating a class based state machine is a safe systemization approach to create a signal dispatcher. Here two signals named *receiveSDU* and *sendPDU* which are defined in *PDCPInterface* and *MACInterface* respectively, are caught by class based state machine and the relevant code to trigger the respecting events are placed here.

The instance based state machine of class *Verifier* contained all the functionalities as a test case. Although in the first iteration the test case is quite simple, but a general pattern for the testing is developed. This pattern includes the following actions (for more information see section *8.3. Blackbox testing VS Gray box*):

- Generating various signals in *Generation* state
- Validating incoming signals in *Validation* state
- Specifying correct or wrong behavior of the system in *Correct* or *Erroneous* states.

RLC component consists of three different subsystems which are representing different modes of RLC. As mentioned before, this case study only focuses on AM mode. Within AM subsystem a class diagram defined, which contains two classes (see Figure 5.3.1.5).

- RLC
- Random

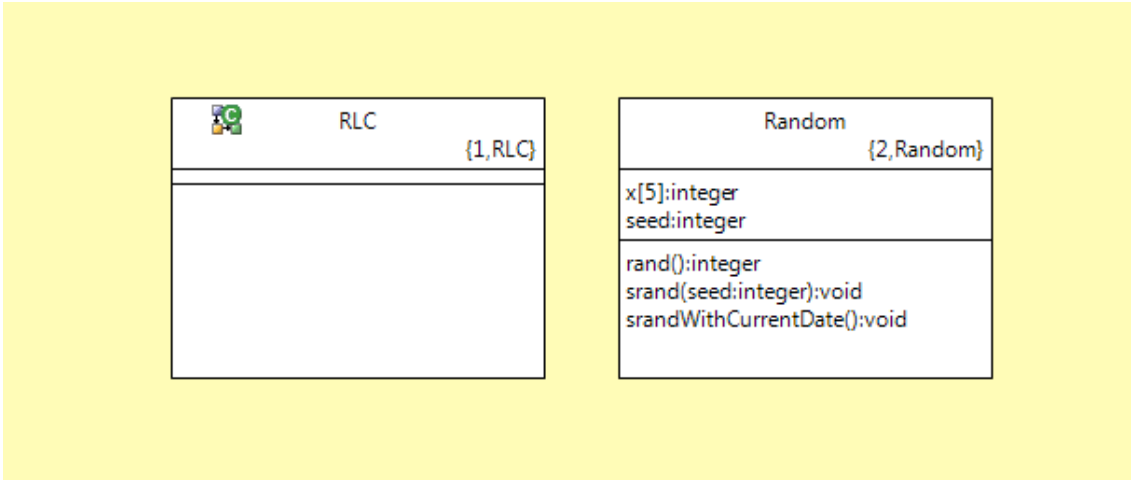


Figure 5.3.1.5 - RLC (AM) class diagram

The RLC class only has a class based state machine as shown in figure 5.3.1.5 by class based state machine's icon. By applying this design decision there is no need to make another class based state machine to handle incoming signals. RLC's state machine (Figure 5.3.1.6) contains two simple states. The *RLC* state is a starting point for the RLC, where RLC waits for an incoming signal from test case (i.e. other layers which are encompassed by *TestFramework*).

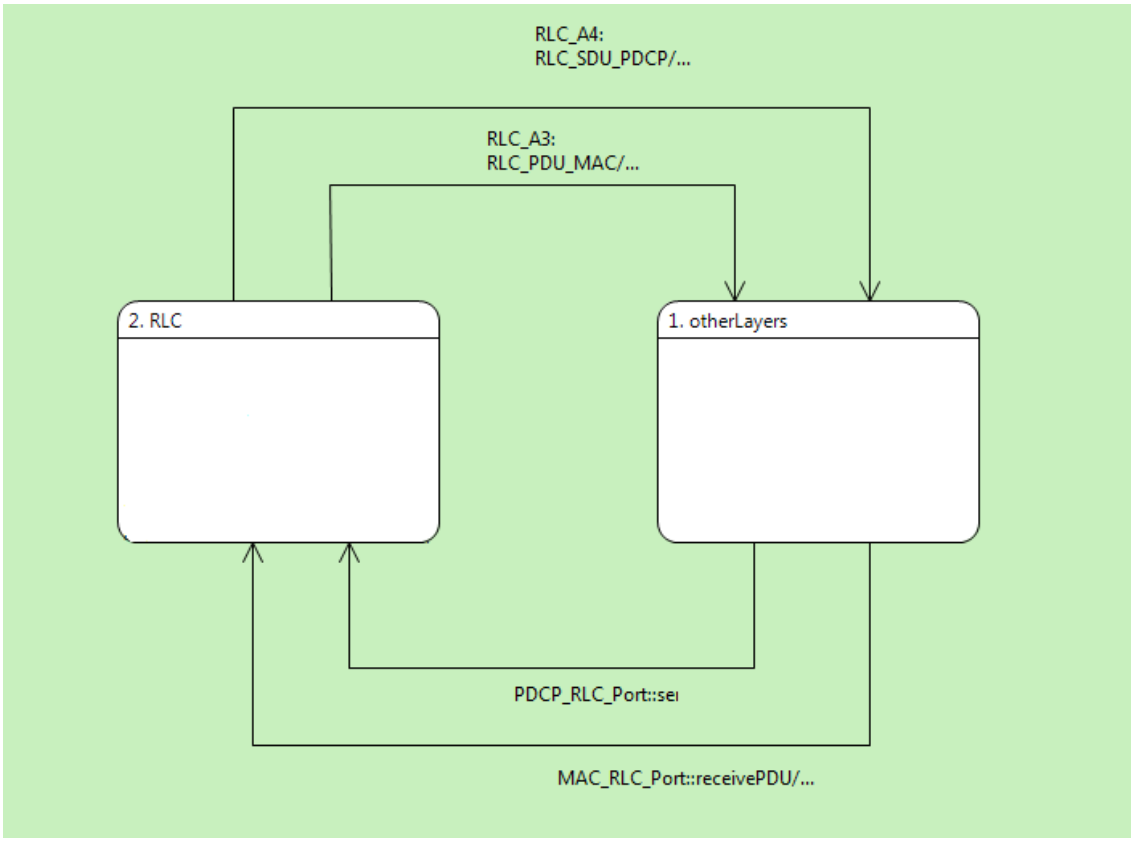


Figure 5.3.1.6 - RLC class based state machine

When RLC got the signals from the test case, a transition bound to the incoming signal makes RLC to pass from *RLC* state into the o*therLayers* state. In this state RLC use random functions of *Random* class to choose between different events. Each event sends a signal back to the test case through the proper interface. No matter which event has been chosen by the random function, RLC ends up in *RLC* state, where it is ready for a next incoming signal.

Inside the *TestFramework* component, besides *TestCases* subsystem another package is placed called *Initialization*. This package is from type *Functions* that may include various global functions. An *init()* function is defined to make the whole system runnable and it is responsible for creating the test case by calling creation event of class *Verifier*. When an instance of class *Verifier* is created, test case generates outgoing signals inside *Generate* state (see Figure 5.3.1.7).

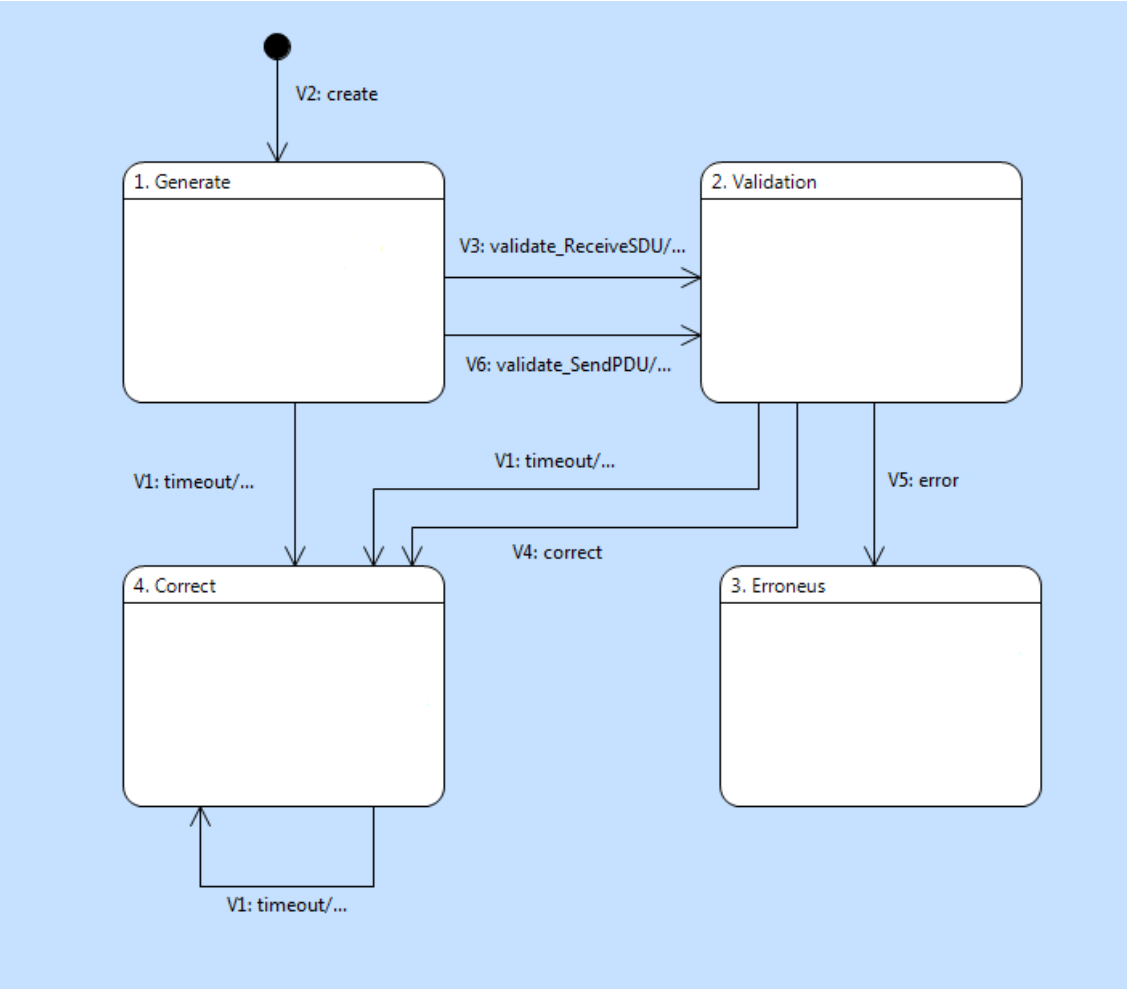


Figure 5.3.1.7 - Verifier instance based state machine

In this iteration with the help of a random function *sendSDU()* or *rececivePDU()* signals were sent to RLC and also a timer has been set. The only possible way for the system to show a wrong behavior is to send an asynchronous signal. This validation will be done in the *Validation* state. By an asynchronous signal we mean getting an unexpected signal from the RLC. Hence the test case will transit to the *Erroneous* state.

The *Correct* state is the place for deciding the number of times that the test case must be run until designers gain confidence on the system's correct functionality.

The test case successfully completes after particular amount of runs determined by checking the variable *counter* in *Tracker* class (see Figure 5.3.1.3). The test case will instantiate another object of itself (i.e. another instance of the test case state machine) until *counter* reaches a predefined value (e.g. 100).

5.3.2. Iteration 2

Duration: 2 Days

Goal: Detecting the right sequence of signals

The second iteration is started by creating the test case based on the pattern that was implemented before. This time test case generates the following signals to the RLC:

- sendSDU() - Through PDCP interface
- receivePDU() - Through MAC interface

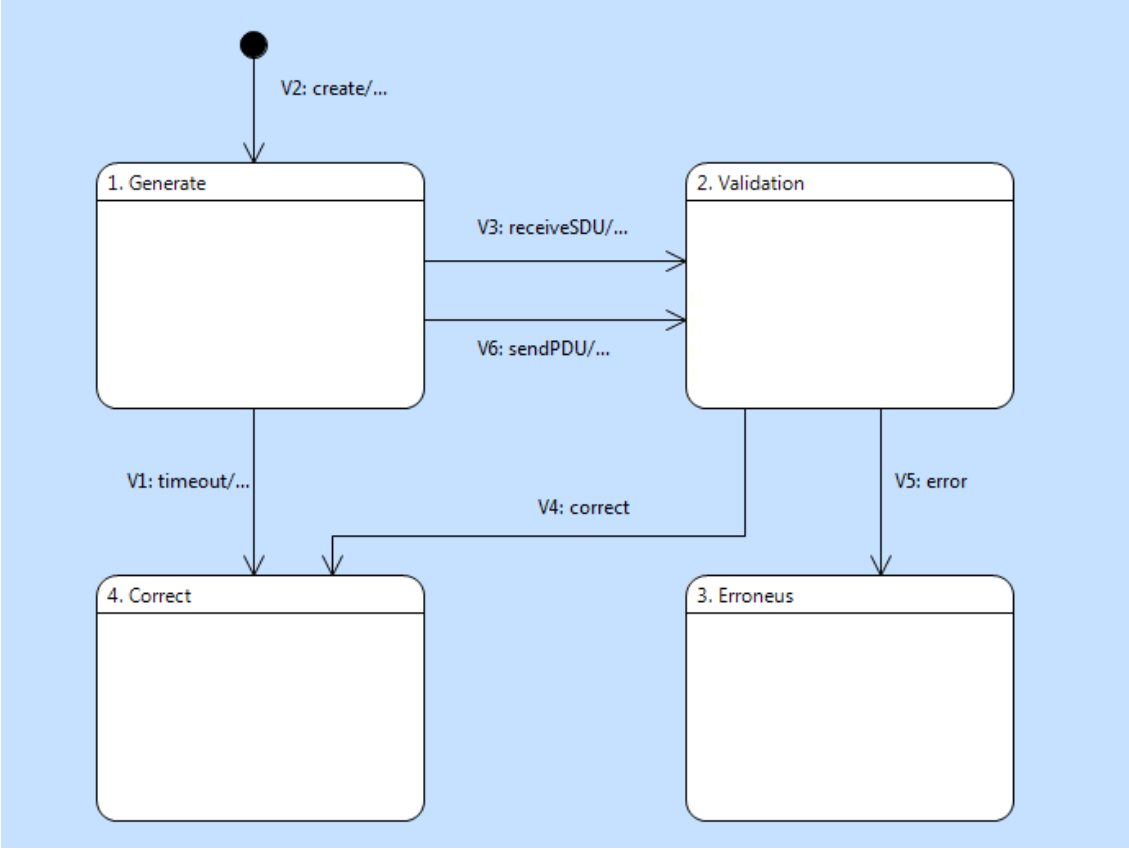


Figure 5.3.2.1 - Verifier instance based state machine

And by setting the timer, test case waits for the RLC component to answer. On the other component, as shown in Figure 5.3.2.2, RLC starts working only when it is on the *Idle* state. By receiving the incoming signals (from test case) which are bound to outgoing transitions, the RLC reaches *Sending* state. Here is the place that RLC randomly choose between three different actions:

- Sending SDU to upper layer (PDCP)
- Sending PDU to lower layer (MAC)
- Doing nothing and go back to *Idle* state

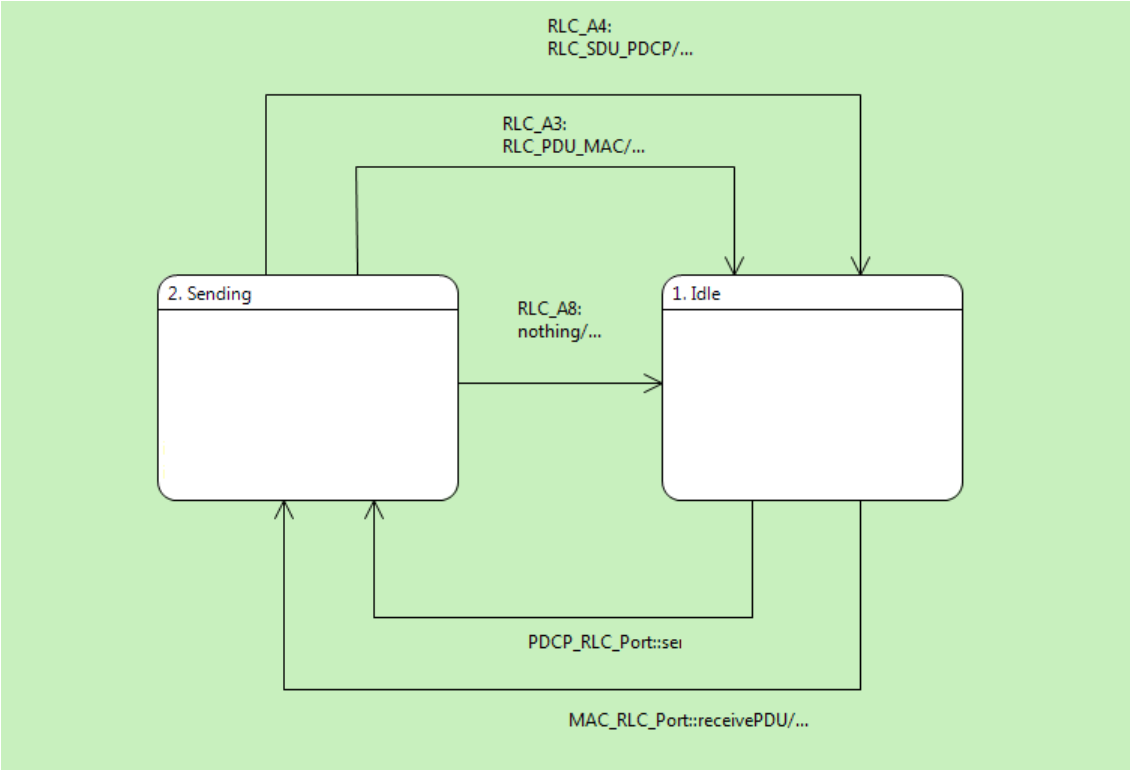


Figure 5.3.2.2 - Test case class based state machine

The first two actions are tied to transitions which send signals *receiveSDU()* and *sendPDU()* back to test case and arrive in *Idle* state. The third action is a transition called “nothing” that represents sending no signal and go back to *Idle* state.

Recall that test case is still in the *Generate* state and it can exit from that state and go further by either getting signals (sendPDU or receiveSDU) or wait until the timer expires and consequently the *timeout* event raised (Figure 5.3.2.1).

In this iteration when the timer is expired, test case goes to the *Correct* state and logically it means that RLC did not answer with a signal and has done nothing. This is a correct behavior of the system when RLC should not send back any signal (e.g. when RLC buffer something and does not need to send any signal).

On the other hand if RLC has sent back a signal, the test case has to detect whether the caught signal was the one that had to be received or not. Therefore with each signal that test case receives, it goes to *Validation* state and checks the incoming signal. The validation performed by comparing values that stored in *signalID* and *catchSignalID* variables. Basically s*ignalID* variable stores the signal number that test case has sent to RLC and *catchSignalID* stores the signal number that test case has received.

By assigning an ID to a signal, test case becomes aware of the right sequence of signals that have arrived in RLC, and also the ones that are extracted from RLC. In other

words, RLC had to answer the incoming signal *sendSDU* only with sending the outgoing signal *sendPDU* and the same holds for incoming signal *receivePDU* which must be answered only by *receiveSDU* signal. If the wrong sequence has been chosen randomly then test case would end up in *Erroneous* state, otherwise it goes to the *Correct* state.

Note: The test case might have separate validation states for different signals that it receives from the target component (i.e. the component under test). In this iteration there is only one validation state due to the simplicity of the system. This approach led to define a variable *catchSignalID* that would be assigned on the transition. Therefore, some operations are hidden in the code and not visible in the state machine.

Recommendation: Always try to trade off between hiding functionalities behind the code and making them visible via state machines. To do so, one may ask questions like, how simple is it to understand the specification whether using graphical images or textual codes. Bear in mind that, if a designer choose more graphics, when the system sends many signals back to the test case then one state for each signal must be drawn. This may cause the following problems when producing a graphical specification (i.e. state machines) for a large system:

- Specification has lots of states which are hard to follow and is not human readable.
- Specification is not easily printable or representable.

On the other hand by hiding operations the following problems may emerge:

- The specification is not easily understandable by various stakeholders.
- To get a quick view of what is the goal on each iteration or how the system fulfills the goals, one might be forced to follow the code on every state or transition.

5.3.3. Iteration 3

Duration: 3 Days

Goal: Detecting the packet loss within the RLC

The packet loss within physical layer is a normal occurrence that might happen in real systems. Therefore the RLC system has to detect the packet loss and take a proper action.

The third iteration began with evolving the test case in order to continuously expand the test framework and suggest a proper pattern for testing systems on high abstractions.

In this iteration, as shown in Figure 5.3.3.1, the test case (i.e. *Verifier*) is divided into two different test case classes, named *Receiver_TestCase* and *Transmitter_TestCase*. Doing so has the following advantages for designers:

- The test case is much more object-oriented.

- It is easier to focus on different activities of the system one at a time. (e.g. in the RLC case study, RLC's transmitter side functionalities can be separated from the RLC's receiver side).

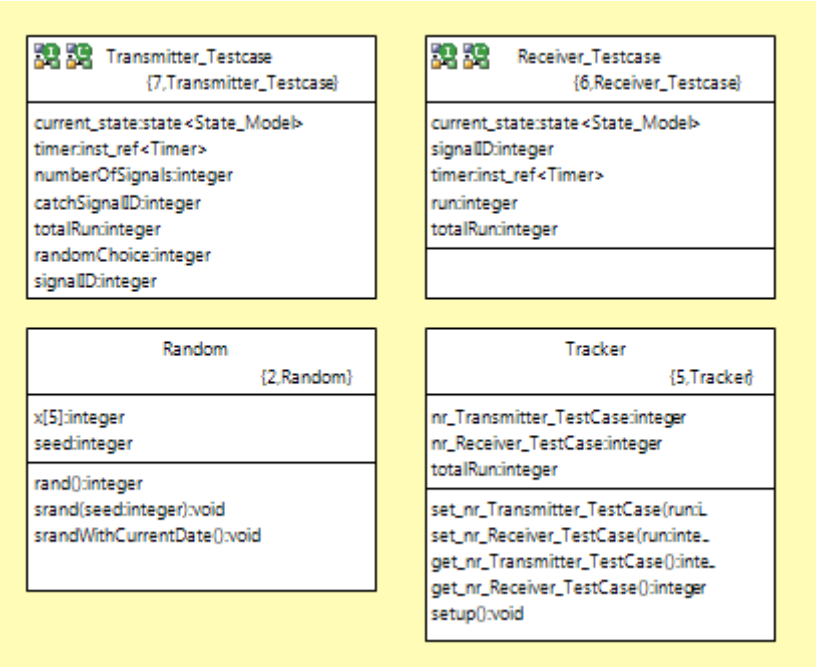


Figure 5.3.3.1 - TestCases class diagram

The mentioned approach might be applicable for different systems where the main functionalities of the system could be separated.

Now that the test case tests RLC separately as a transmitter side and a receiver side, only *sendSDU()* signal could be sent from *Transmitter_TestCase*. Accordingly the only signal that is sent back to the test case from RLC is *sendPDU(id)*. The same stands for *Receiver_TestCase*, therefore the only outgoing and incoming signals are *receivePDU(id)* and *receiveSDU()*.

Notice that MAC interface signals, *sendPDU(id)* and *receivePDU(id)* have a parameter called *id*. This parameter is defined to detect the packet numbers (i.e. packet IDs).

Consider the *Transmitter_TestCase* (see Figure 5.3.3.2), as mentioned before the only outgoing signal from this test case is *sendSDU()* therefore in the *"Generate"* state the test case randomly chooses whether send this signal or not, and set the timer as well.

If the timer expires then the test case goes to *Validation* state (i.e. timeout validation) by *timeout* event and test case must validate two different scenarios. One scenario is that, in *Generate* state if the test case has chosen sending no signal then it is a normal behavior of the system to answer with no signal, hence the test case ends up in *Pass* state. On the other scenario, if the test case has chosen *sendSDU()* signal, then time out in the test case shows that there is something wrong with RLC that could not send back *sendPDU(id)* signal, and maybe it has crashed, hence the test case ends up in *Fail* state. Notice that if the test case wants to validate the time out there is no way to do so unless

remembering the generated action (here *randomChoice* is a variable defined to store the generated action).

When the test case got a *sendPDU(id)* signal back then in the validation step it checks the *id* parameter and if it matches to what was expected from the RLC, then the test case ends up in the *Pass* state and contradictory for the *Fail* state.

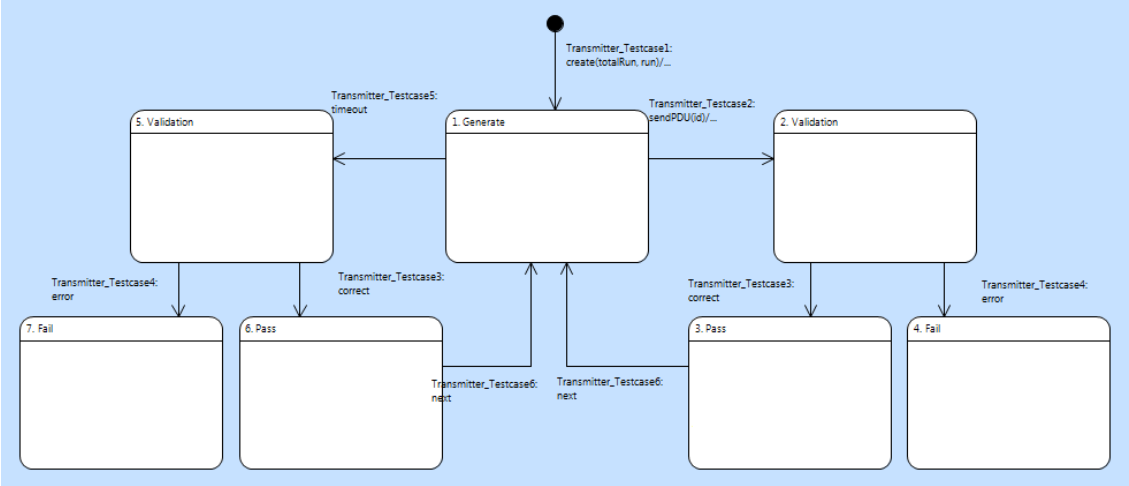


Figure 5.3.3.2 - Transmitter_TestCase instance based state machine

As shown in Figure 5.3.3.3 RLC's state machine is expanded from two states into four states.

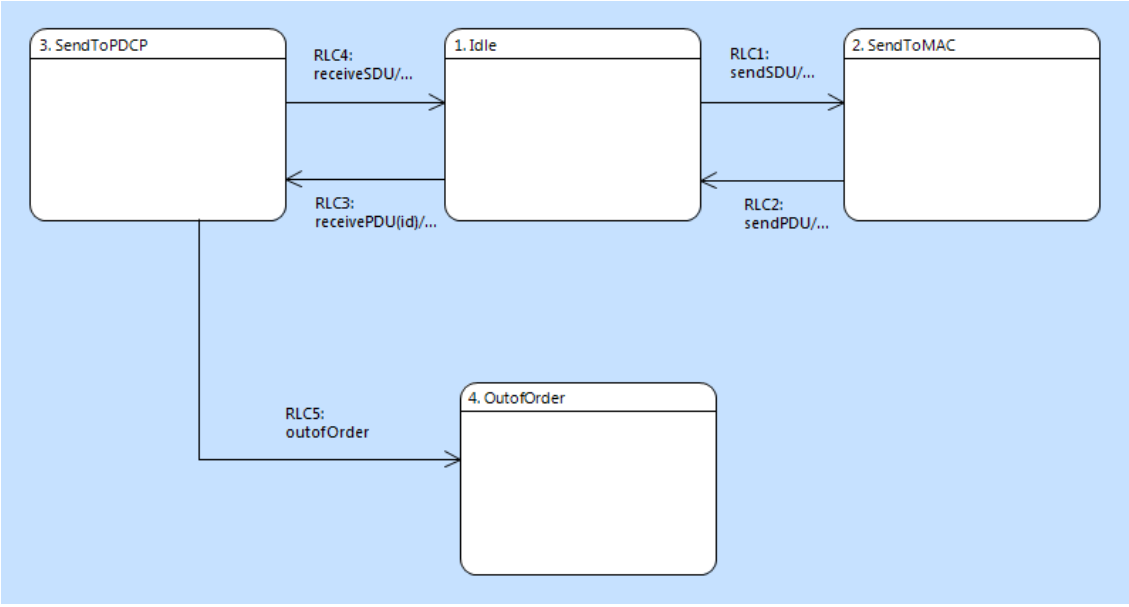


Figure 5.3.3.3 - RLC class based state machine

As before, RLC starts to operate from the *Idle* state but this time for different signals that has been received it goes to different states. If the signal has been sent by the *Transmitter_TestCase* (i.e. *sendSDU()*), then RLC goes to *SendToMAC* state where it sends sendPDU(id) signal back to the test case. A counter assigned to the id parameter that stands for the packet number. Thus for each incoming signal, packets in sequence

would be sent back to the test case. On the other hand if the RLC got a *receivePDU(id)* from *Receiver_TestCase* it would pass through *SendToPDCP* state and by checking the *id* parameter takes the proper action. For expected packets RLC sends a *receiveSDU* signal and for unexpected ones it ends up in the *OurOfOrder* state.

The *Receiver_TestCase* is designed with a slightly different pattern that is illustrated in Figure 5.3.3.4. The *Validation* states are omitted and the correct and wrong outgoing signals are categorized in two different types called *expected* and *unexpected*. The *Generate* state is responsible for generating correct or expected order of the packet IDs; which are incremented by one each time the signal has been sent and also an unexpected order of the package IDs which are simply incremented by two.

Instead of the *Validation* states, transitions namely, *expected()* and *unexpected()* are leading the test case to the states *Expected_Waiting* and *Unexpected_Waiting* respectively. The test case waits there until it gets a time out or a *receiveSDU* signal. If the test case has generated expected behavior (i.e. a correct order of the packets) and waiting for *receiveSDU* signal, then by catching that signal it goes to the *Pass* state and with a time out it ends up in the *Fail* state. That is because RLC must send back a *receiveSDU* right away after it gets the expected packet, otherwise the RLC might has crashed. On the other hand by generating unexpected packets RLC will find out that the packet is not the expected one therefore it goes to the *OutOfOrder* state and stay there until the timer of the test case expires. Hence if the test case got time out for unexpected packets it is a correct behavior of the system.

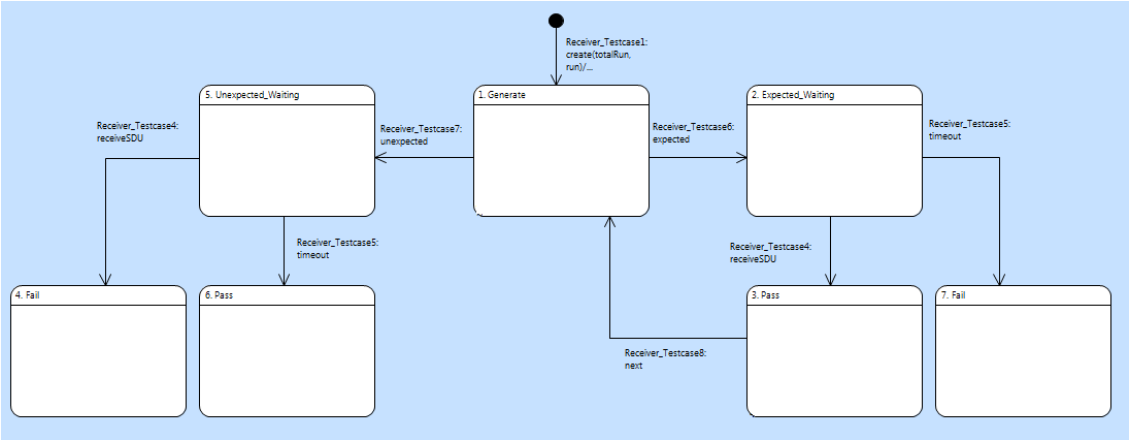


Figure 5.3.3.4 - Receiver_TestCase instance based state machine

Recommendation 1: The latter pattern for *Receiver_TestCase* might not always be applicable, because in many situations unwanted data or behavior of the system could not be distinguished easily; furthermore this pattern might lead to unnecessary states that are not doing anything (e.g. *Expected_Waiting* in figure 5.3.3.4).

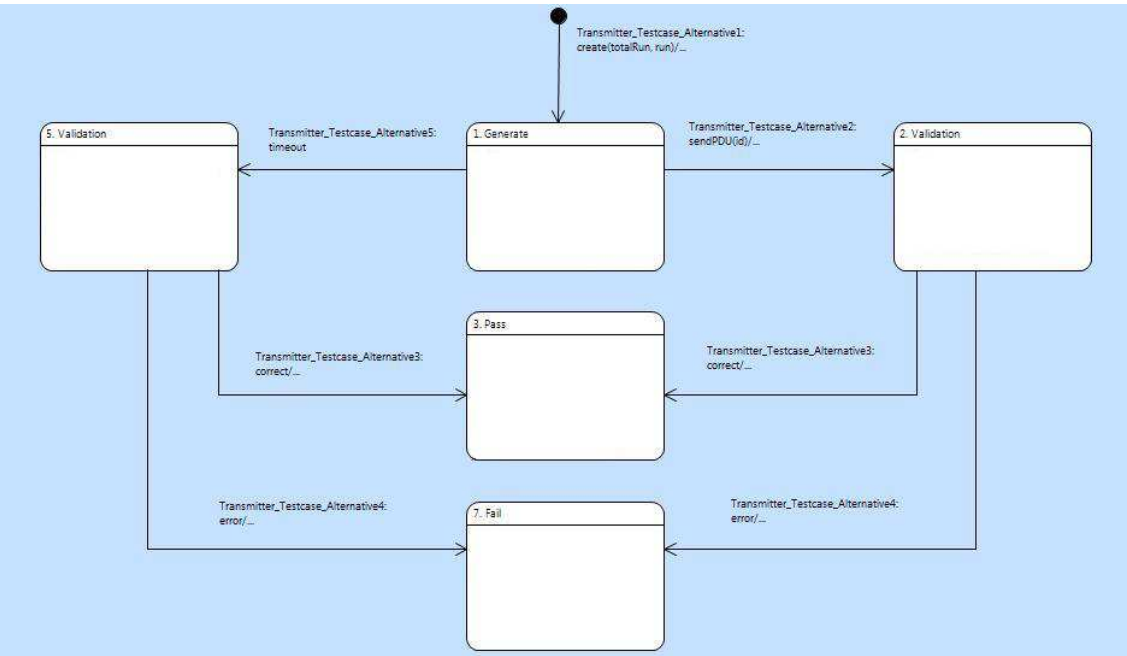


Figure 5.3.3.5 - Alternative “Transmitter_TestCase” instance based state machine

Recommendation 2: In *Transmitter_TestCase*, states with the identical operations, like different *Pass* and *Fail* states could be merged into exactly one *Pass* and one *Fail* state as shown in Figure 5.3.3.5. If some information or logging needed for different passes or fails, that information could be placed on the transitions.

Note 1: Name of states *Correct* and *Erroneous* have changed to the *Pass* and *Fail* respectively. The former *Correct* and *Erroneous* states were representing the correct behavior of the system. Even though the test case ended up in *Erroneous* state the error represented the correct behavior of the RLC according to the specification and the reason for the error was feeding the system with wrong data. Hence from this iteration the *Pass* state contained all the correct behaviors of the system including the correct behavior of a system for unwanted data generated by the test case. The *Fail* state represents the actual failure of the system.

Note 2: In this iteration the test case was not recurred by creating a new instance of the test case, and it loops around *Pass* to *Generate* states via the *next()* transition (see Figure 5.3.3.2 and compare it with figure 5.3.2.1). The reason behind this systematic change was that if a new attribute or counter needed to be defined and had to store a value then by creating a new instance of the test case the stored data would be lost (i.e. assigned to the default value).

5.3.4. Iteration 4

Duration: 5 Days

Goal: Re-ordering packets

This iteration focused on the receiver side of the RLC. As before, iteration begins by developing the test framework and the *Receiver_TestCase* extended by keeping the

introduced pattern. The goal is to reorder packets that sent by the test case as PDUs. To simulate the packet loss through physical layer, a hard coded string (e.g. HELLO) was defined in the test case and each letter is chosen randomly by the test case and sent as a PDU through a *sendPDU(id, char)* signal. The *id* parameter is the same variable that was defined in previous iterations and *char* is a new parameter that transfers the letters to the RLC component. As mentioned above these letters will be chosen randomly thus the test case might send letters out of order which represents the natural behavior of the physical layer. On the other side the receiver of these letters must catch and re-order them, based on the packet IDs. Test case runs until all the letters sent by test case and also received back from the RLC in a correct order, therefore two Boolean arrays with the exact same size as the hard coded string array of letters was defined. Array named *sent* keeps track of the sent letters by setting their positions to *true*, and the other for receiving letters called *received*. The test is completed when all elements of both *sent* and *received* arrays are assigned to *true*.

After generating *sendPDU(id, char)* with a random letter and id as the parameters in *Generate* state, test case waits until either the timer expires or receive a signal from the test case (Figure 5.3.4.3).

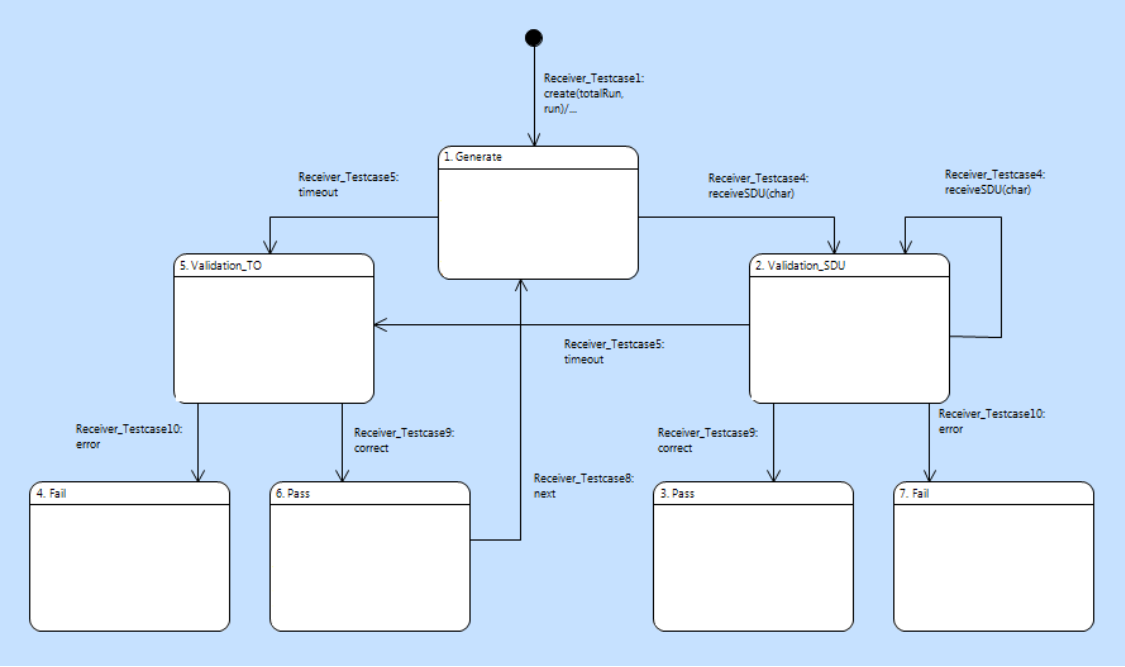


Figure 5.3.4.3 - Receiver_TestCase instance based state machine

As you can see in Figure 5.3.4.2, when the RLC receives a *sendPDU(id, char)* signal first it goes to the *Buffering* state. This state is responsible to check if the receiving packet was the expected one or not, based on the packet id. If the packet was unexpected then the RLC must buffer the packet (if it was not already buffered) and go back to *Idle* state where it waits for other packets. Notice that besides the extended functionality and new extra states, there are more attributes defined in the RLC's class (Figure 5.3.4.1). For instance to form the buffer in RLC, two arrays were defined in

RLC one for buffering letters and one for the packet ids. Because each id is bound to one letter, both arrays are indexed by a same pointer called *bufferIndex*.

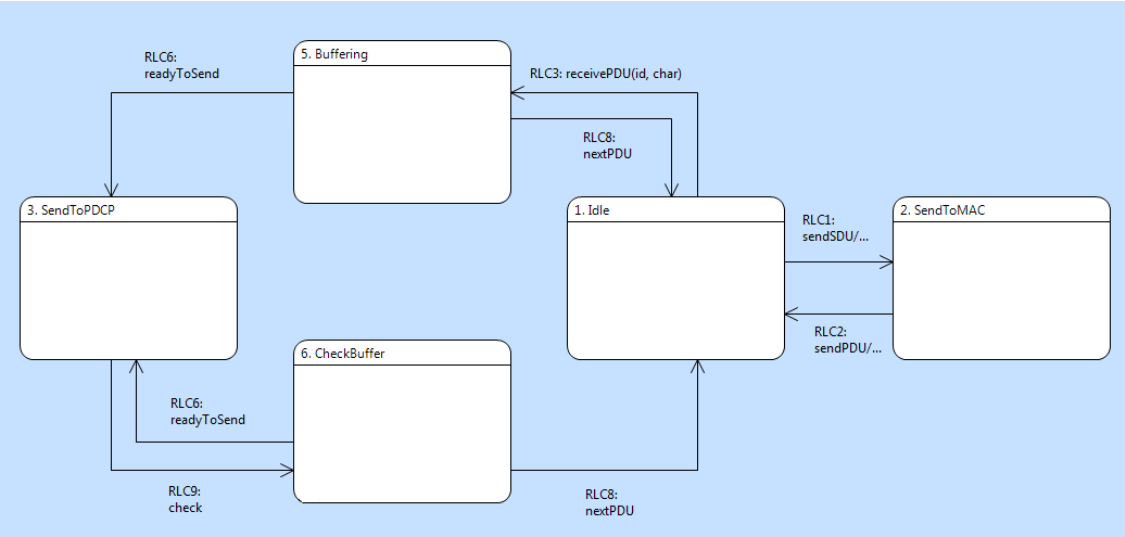


Figure 5.3.4.2 - RLC instance based state machine

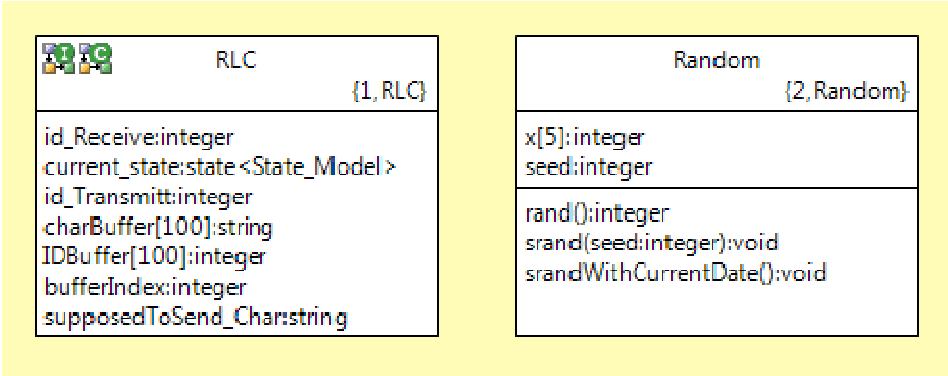


Figure 5.3.4.1 - RLC class diagram

If the packet was the expected one, the RLC goes to the *SendToPDCP* state that has the same functionality as it had before. In that state the letter is sent by a *sendSDU(char)* to the test case without buffering the letter. But unlike former iterations the RLC is not directly going back to the *Idle* state. Here RLC first checks that if there are any other buffered packets which must be sent due to the fact that they are the expected ids. Thus the loop between *SendToPDCP* and *CheckBuffer* states. If there was no packet with the current expected id in the buffer then RLC goes to the *Idle* state and waits for other *sendPDU(id, char)* signals.

After sending packets in the *Receiver_TestCase* if a *receiveSDU(char)* sent back from the RLC then the test case validates that whether the incoming letter received in a correct order. This is done by comparing the *char* parameter with the *next* character that the *Receiver_TestCase* supposed to receive. If the comparison failed it means that RLC didn't sent back the correct character and failed to re-order packets. Consequently the test case would end up in the *Fail* state. Opposite to the former test case the succeeded

comparison in *Validation_SDU* does not lead to the *Pass* state. The reason is, there might be other packets in the RLCs buffer that are sent by more *receiveSDU(char)* signals and should be validated in the same state before ending up in the *Pass* state. Therefore if the validation of the signal succeeded the test case starts a timer to get all *receiveSDU* signals. After receiving all signals and comparing all incoming letters, the timer expires and with a *timeout* event test case goes to the *Validation_TO*. In that state, test case must consider why the timer has been expired. If the test case has sent a letter through *sendPDU(char)* in the wrong order, then getting a timeout is a correct behavior of the system and that means RLC caught the packet, buffered it and went back to the *Idle* state where it is waiting for the next signal. Diversely if the test case has sent letters in a correct order and got a time out then RLC did not send back those packets that needed no re-ordering. Based on the time out validation, test case ends up in the *Fail* or *Pass* states.

Notes 1: In this iteration names of the validation states (Figure 5.3.4.3, *Receiver_TestCase*) changed to *Validation_SDU* and *Validation_TO*. This changing of the state's name was made to clarify different validation actions in each state. It also helps the *BridgePoint* debugger. (see 8.4)

Notes 2: The semantics of the loop on *Validation_SDU* through *receiveSDU(char)* transition could be implemented through a loop between the *Pass* state and the *Validation_SDU* but then the time out validation might alter as well.

5.3.5. Iteration 5

Duration: 3 Days

Goal: Status packets (ACK/NACK)

In the former iteration, RLC succeeded to receive PDU packets from a noisy environment (Simulated by the test case) and reorder them in case that they were out of order.

The only functionality that added to the RLC in the current iteration was sending status packets (i.e. ACK or NACK) back to the test case. As shown in Figure 5.3.5.1, the structure of the state machine is not altered at all and the mentioned functionality is appended to the *nextPDU* transitions. While RLC buffered a packet and is ready to go to the *Idle* state, a NACK packet will be sent to test case as well. Sending a NACK is specified by a *status(ack, id)* signal. Parameter *ack* defined as a boolean type and False value indicates on a NACK packet. The *id* parameter shows the packet number that the status has created for. Whenever RLC succeeded to send SDU packets an ACK status will be created and sent to the test case. The *id* parameter for ACK status packets indicates that, all PDUs up to that particular id has been received and corresponding SDUs has been sent back.

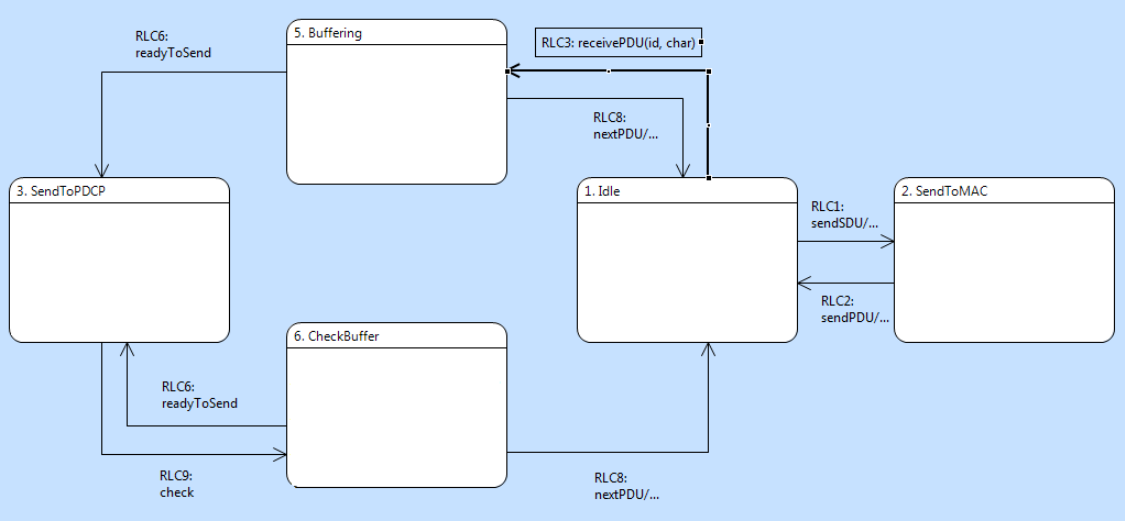


Figure 5.3.5.1 - RLC instance based state machine

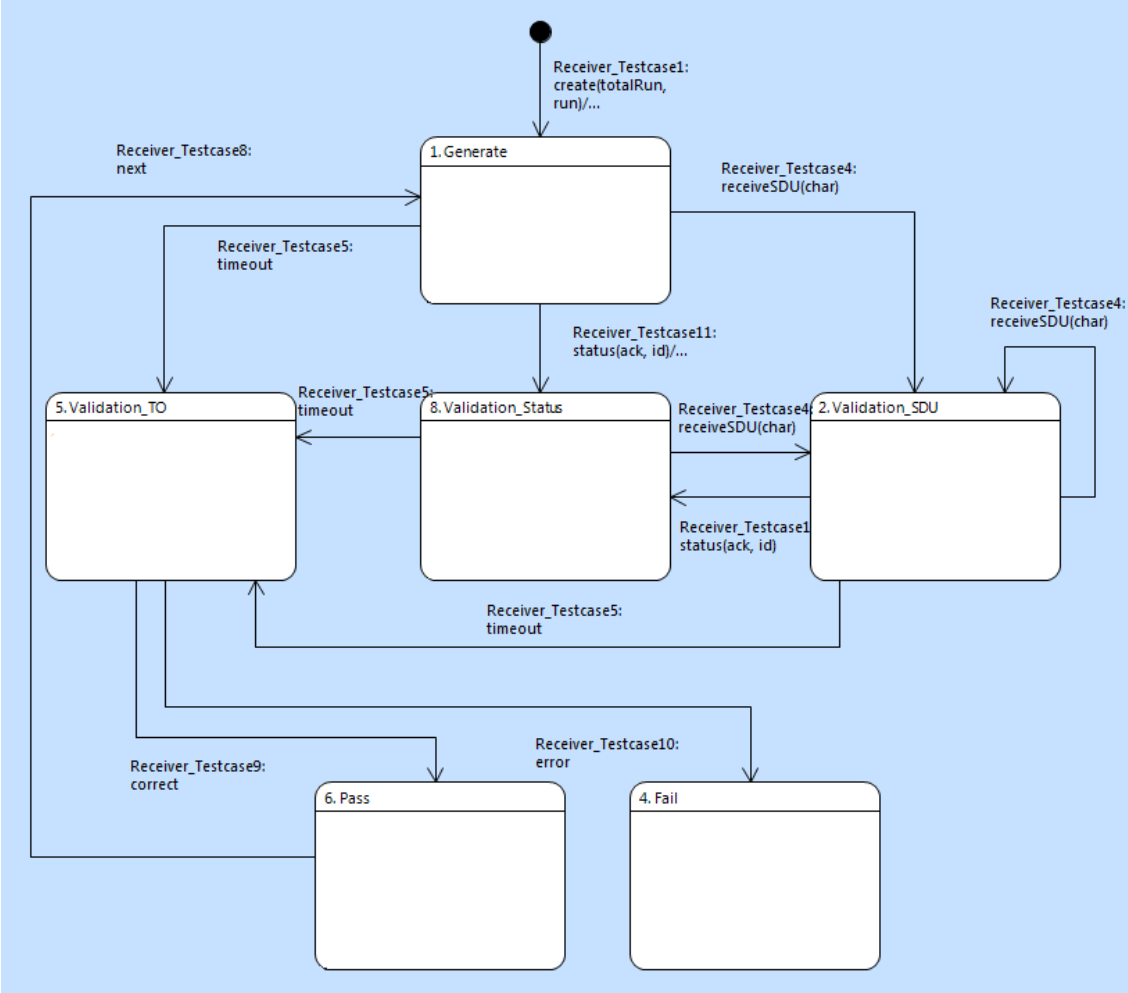


Figure 5.3.5.2 - Receiver_TestCase instance based state machine

As illustrated in Figure 5.3.5.2, the *Receiver_TestCase* has changed due to the fact that there are two incoming signals back to the test case. Bear in mind that the focus is on the *Receiver_TestCase* because the receiving side of RLC has been changed.

The *Generate* state remained unchanged and the same signals are produced within that state. As mentioned before if the pattern of the test case intended to be kept then for each incoming signal a validation state must be added. Thus an extra state has been added to validate the incoming *status (ack, id)* signal.

Note 1: In the recommended test case pattern, a *Validation* state must be added for each incoming signal.

Since status packets are received from the MAC layer and passed through a noisy physical layer, the test case could not be confident that status packets are related to incoming SDUs. Another issue that may arise is that packets might be transformed into wrong packets (e.g. NACK 2 status packet might turn into ACK 3). Therefore in some situations receiving status packets are not corresponding to receiving SDUs. In this view test case would not determine the pass and failure of a signal without validating related signals. Hence from the *Validation_SDU* a forwarding transition has defined to validate the status as well. One problem that is still remaining is the order of incoming signals. Thus, the same semantics applied to *Validation_SDU* can be applied to *Validation_Status* as well. Finally when both signals are validated, the *Validation_TO* (i.e. time out validation) makes the final decision about failure or correctness of the system.

Notes 2: When the order of incoming signals cannot be determined, forwarding transitions should be added among concurrent validation states, to handle and visualize the concurrency (Figure 5.3.5.2 shows the concurrency between status and SDU packets).

5.3.6. Iteration 6

Duration: 3 Days

Goal: Re-transmission

As mentioned before the only functionality that was added to RLC was sending ACK or NACK status back to the test case from the receiver side of RLC. This iteration focused on the transmitter side of RLC and the goal was that for each status signal that RLC (the transmitter side of RLC) received, it needs to perform the correct action after analyzing the type of the status. The correct action is, if the test case has sent a NACK for a specific packet, then RLC must re-send the same packet again, otherwise if RLC received an ACK with a specific id, then RLC would be noticed that up to that id all the packets are received.

As before the test case starts by sending signals to RLC within the *Generate* state. This time, test case randomly chooses to send between three different signals:

- *sendSDU(char)* signal: The parameter *char* represents one of the letters in the word ‘HELLO’.

- *statusTXtoRLC(ack: true, id)* signal: In which *ack* parameter is assigned to True, so that it represents ACK status. *id* parameter specify the packet id (e.g. *statusTXtoRLC(ack: true, id: 5)* is an ACK up to id = 5).
- *statusTXtoRLC(ack: false, id)* signal: Which is representing NACK for a specific packet.

The Figure 5.3.6.1 shows the different states of the RLC. If a *sendSDU(char)* has been sent, then RLC goes to *RetransmissionBuffer* state where a copy of the packet (i.e. the letter) will be kept in *charRetransmissionBuffer*. Bear in mind that another boolean buffer called *acknowledgementBuffer* defined within transmitter side of RLC with the exact length of *charRetransmissionBuffer*. This buffer keeps track of packets which RLC got the ACK for them.

After passing through *RetransmissionBuffer* state, RLC sends back a *sendPDU(id, char)* signal to test case. The *char* parameter is the same *char* that has been sent by test case and *id* is assigned by a counter (Because test case sends characters in sequence through *PDCPInterface*).

When a status signal (either ACK or NACK) received, then RLC analyze the status type in a state called *Check_RetransmissionBuffer*. Since test case generates ACK or NACK with random ids, RLC must discard those status packets which have inappropriate ids.

If the status signal was an ACK, RLC checks for the id. if the packets before that id have been sent back by RLC, then receiving an ACK means all packets up to that id correctly received by RLC's peer. Therefore RLC remove those packets from its *retransmissionBuffer*.

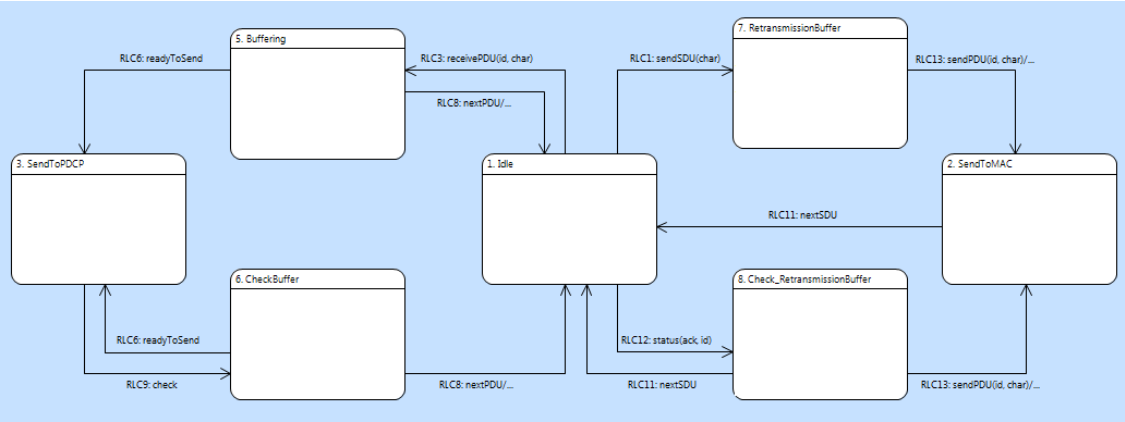


Figure 5.3.6.1 - RLC instance based state machine

On the other hand if the status was a NACK, this time RLC checks two cases to understand whether discard inappropriate NACK or re-send a packet. First it checks if the packet with specified id has been sent before (by checking the counter "next" for packet ids which were sent for the first time) and also it checks whether ACK for that particular id has been received already or not (by checking the *acknowledgementBuffer*).

In case of discarding the status, RLC goes back to the *Idle* state and waits for the next status or SDU signal, but if there was any proper NACK, RLC re-sends the packet passing through the *SendToMAC* state.

According to the test framework pattern, *Transmitter_TestCase* waits for a signal from RLC or a time out. Different states of the test case are depicted in Figure 5.3.6.2. If the test case got a *sendPDU(id, char)* then within the *ValidationPDU* state, it checks whether status signal (i.e. ACK or NACK) or a SDU packet has been sent before. In case of a SDU packet, test case checks the letter that has been sent and the *char* parameter that got back from RLC, and ends up to the *Pass* state if they were equal (vice versa for the *Fail* state). In case of ACK signal, the test case directly goes to the *Fail* state. This means that test case has sent an ACK and RLC responded by sending back a PDU which is a wrong behavior of RLC. In case of NACK signal, the test case has to know about the functionality within RLC to decide a correct or wrong behavior. The correct functionality of RLC is it must check the *acknowledgementBuffer* before re-sending a packet. Therefore test case needs to keep track of *acknowledgementBuffer* by defining an identical array called *shadowAcknowledgementBuffer*. This shadow buffer is being maintained whenever the test case generates an appropriate ACK in the *Generate* state. Using this shadow buffer, the test case first checks that if the packet has been sent before and also no ACK for that particular packet received by RLC. Secondly test case checks that whether re-sent *char* from RLC is equal to the letter that test case expected to be sent back. If all these conditions hold then test case goes to the *Pass* state, otherwise it will end up in the *Fail* state.

When the timer expired after *Generate* state, if an ACK status has been sent then time out is a correct behavior of the RLC. This is because, the RLC either has discarded an inappropriate ACK or maintained the *acknowledgementBuffer* and went back to the *Idle* state (Figure 5.3.6.1). In case of a NACK status the test case checks the *shadowAcknowledgementBuffer* and if there is nothing to be re-sent then the test case ends up in the *Pass* state but if there is any packet that supposed to be re-sent the test case must go to the *Fail* state (Figure 5.3.6.2).

The test is completed when all the characters are sent as SDU packets, and RLC receives an ACK status for each PDU packet (i.e. when all elements in *shadowAcknowledgementBuffer* array are assigned to True).

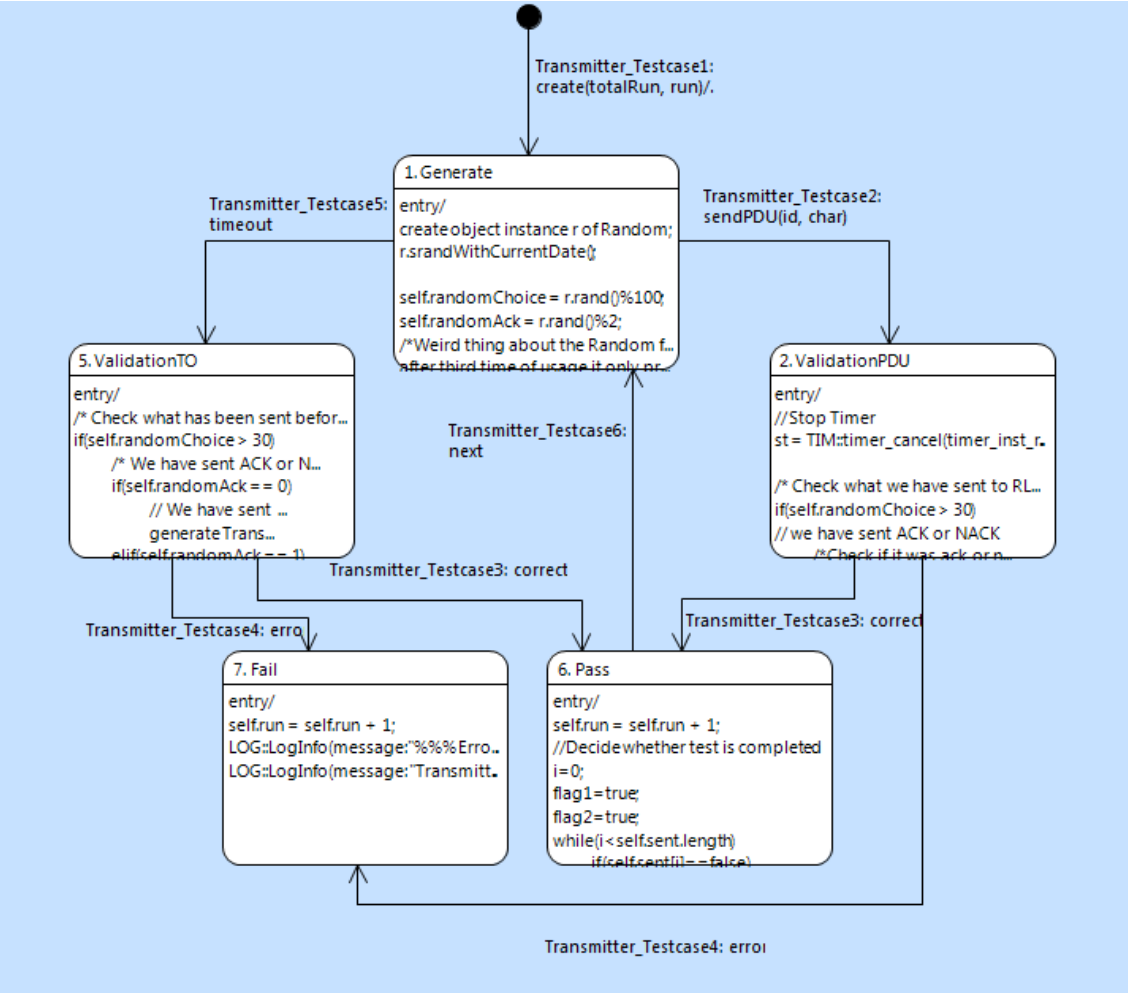


Figure 5.3.6.2 - Transmitter_TestCase instance based state machine

Note 1: While performing a black box test, the test component has no access to the component under test except through defined interfaces. In some cases the test case might need some functionalities or data structures from the target component. On this point, to use shadow structures is helpful in order to keep track of the system under test. For more information see section 8.3.

5.3.7. Iteration 7

Duration: 2 Days

Goal: Refactoring

During the last iterations some unnecessary attributes and variables were defined for different classes and state machines. Furthermore some crucial design changes must be applied to the system before it grows larger and more complex. Bear in mind that following design changes have not forced new functionalities to the system. Thus RLC kept the same capabilities and refactor them to meet the requirements in a realistic manner.

Figure 5.3.7.1 - RLC and RLC Prime structures

As shown in Figure 5.3.7.1, in a real telecommunication systems there are two RLC layers that communicate with each other (e.g. RLC and RLC Prime).

The designed system up to now has mixed up two different entities from different RLC peers into one RLC. Therefore there was no connection between the receiving side of one RLC and the transmitting side of the same RLC. Figure 5.3.7.1 illustrates the omitted entities within the current system (i.e. dotted sections in different RLC components).

This iteration focused on the mentioned issue, and eliminated the short comes driven from simplified design along with removing avoidable classes, attributes, variables, algorithms and so on.

As shown in RLC's class diagram (Figure 5.3.7.2) RLC class is divided into two parts each representing a different side of the RLC. This solution also is more object-oriented and helps designers to focus on one side, without altering the whole system.

Notice that the divided RLC does not have a class based state machine. Binding of the incoming signals is now done in another class named, *RLC_Dispatcher*.

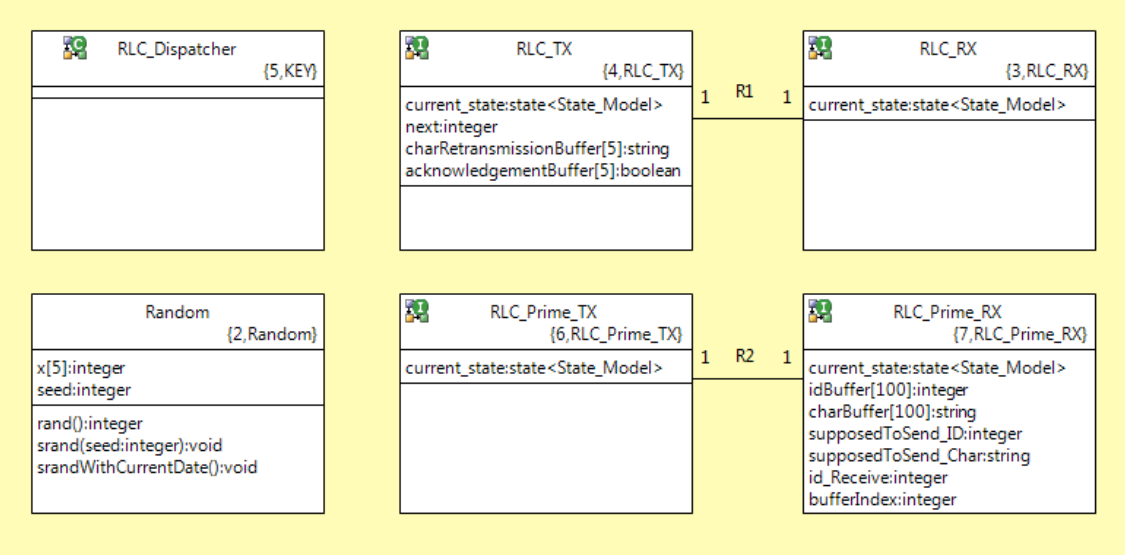


Figure 5.3.7.2 - RLC Class diagram

The last change in class diagram is definition of peer RLC named as *RLC_Prime* class (divided into the mentioned entities).

Now that the both peers are defined in the class diagram, pre-existing state machine for the receiving side is copied into *RLC_Prime_RX* and the transmitting side remained on the *RLC_TX*.

The omitted entities are filled with an instance based state machine that connects two side of the RLC. According to the real RLC system, the transmitter side of the RLC cannot receive any signal from the MAC interface. Also the receiver side cannot transmit any packet to the MAC interface. Therefore another realization requirement injected to the system. From now on, the receiving side is only responsible for the incoming signals from MAC and packets delivered to the transmitter side in case that a transmission is needed. Figure 5.3.7.3 illustrates the actions via state machines. Notice that RLC operates as a transmitter peer and RLC Prime as a receiver peer.

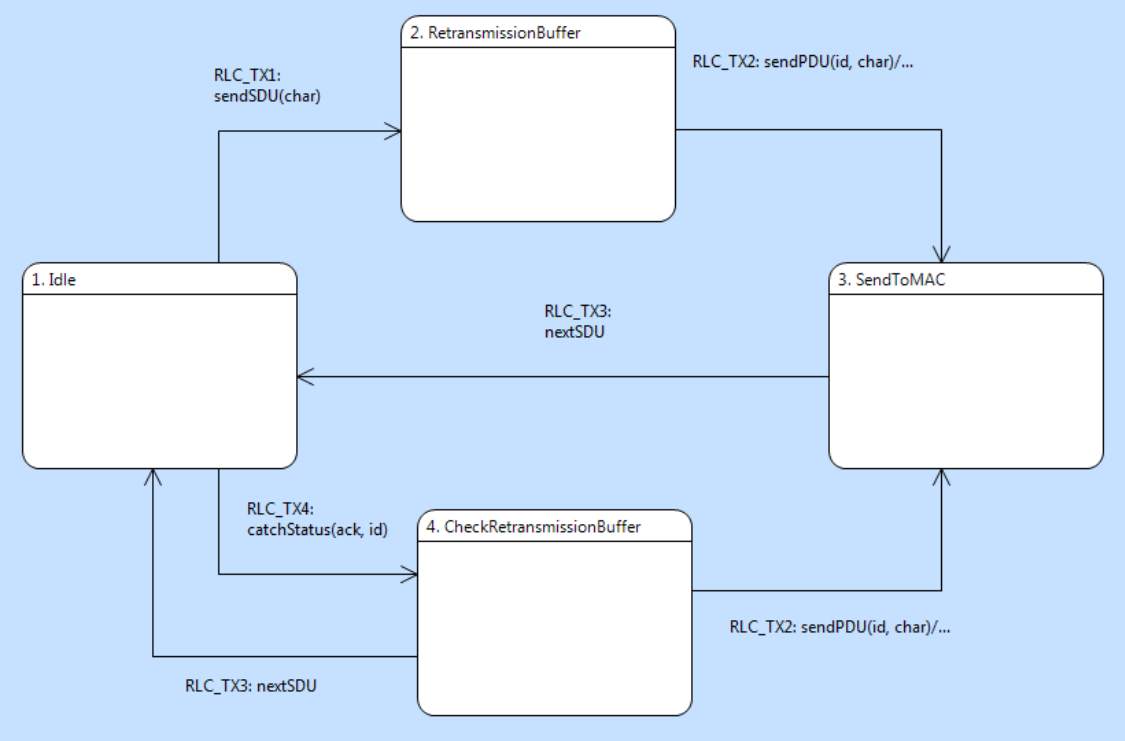


Figure 5.3.7.3 - RLC_TX Instance based state machine

As shown in figure 5.3.7.4, the *RLC_RX* state machine is responsible for buffering the received status packets and delivering them to the TX side.

Figure 5.3.7.6 shows that the *RLC_Prime_TX* is responsible for transmitting status packets which are created by the RX side of the RLC Prime. For more information about refactoring step see 8.2.

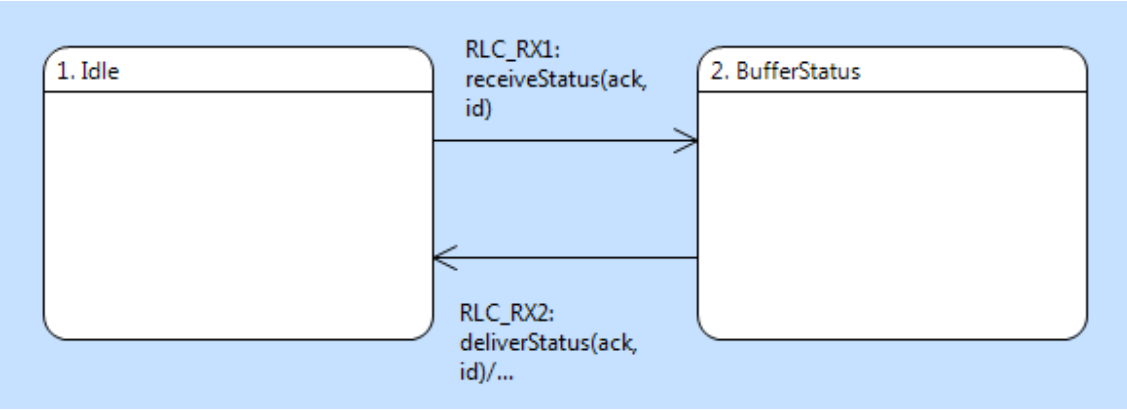


Figure 5.3.7.4 - RLC_RX Instance based state machine

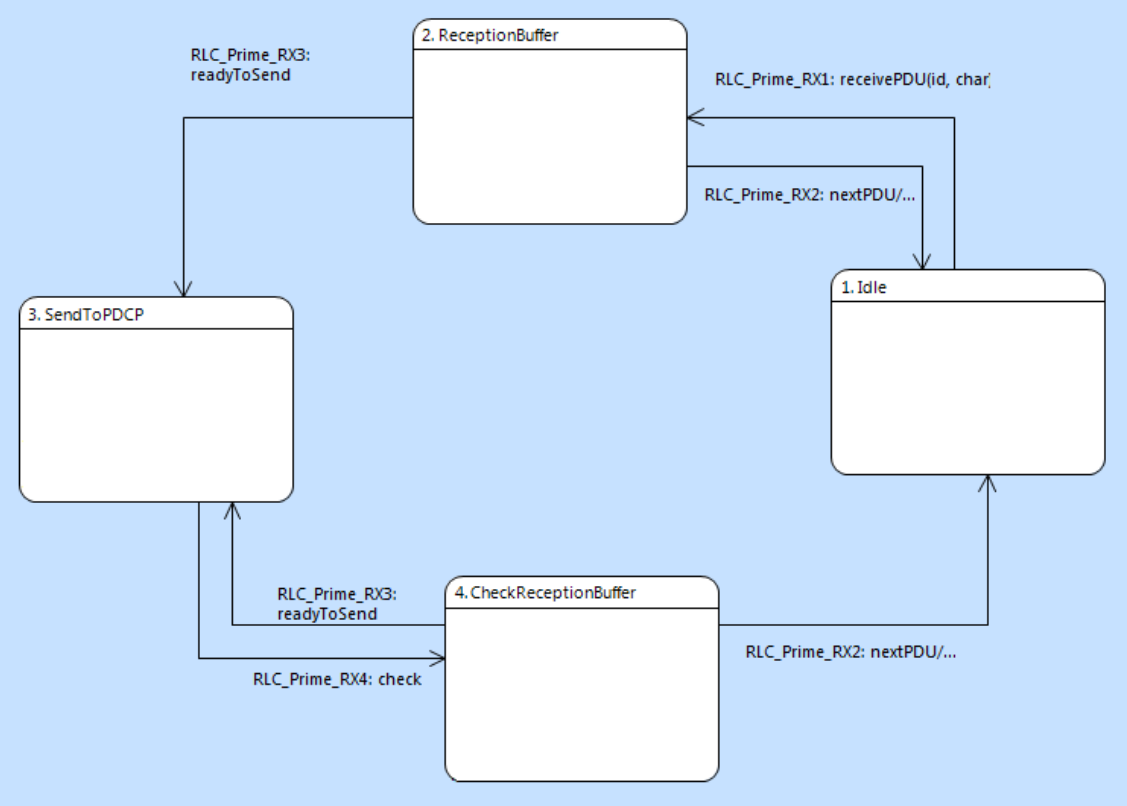


Figure 5.3.7.5 - RLC_Prime_RX Instance based state machine

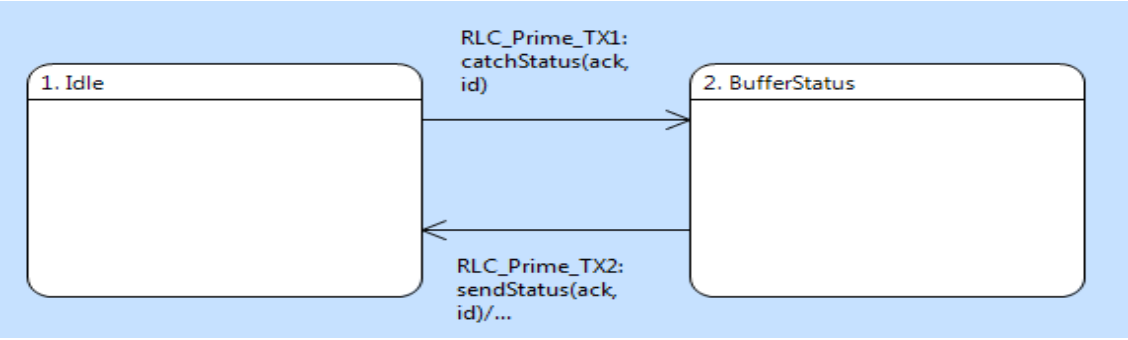


Figure 5.3.7.6 - RLC_Prime_TX Instance based state machine

5.3.8. Iteration 8

Duration: 5 Days

Goal: Transmission opportunity (The receiver role of RLC)

After refactoring the system, designers can focus on adding other functionalities to the system. So far the receiver RLC (bear in mind that, this does not mean the receiving side) and transmitter RLC are divided into two different RLC systems. Each RLC system has two classes called *RX* and *TX* which represent the receiving side and the transmitting side respectively. But neither RX side nor TX side of the two different RLC entities are identical. As a matter of fact, *RX* side of the RLC which has a receiver role is more developed than *RX* side of RLC as a transmitter role. The same goes for the *TX* side of a different RLC with different roles. Along with defining transmission opportunity, this iteration combined two RLCs with different roles into one RLC entity that can handle both roles. Notice the class diagram in Figure 5.3.8.1.

As always the iteration started by extending the test case. Since this iteration focused on the receiver role of the RLC, the *Receiver_TestCase* was responsible for testing the complete RLC.

A receiver entity of RLC is able to receive PDUs from its peer, and if the packets were the expected ones then send SDU to the upper layer (i.e. PDCP) along with making status packets to inform the peer about incoming PDUs. But as formerly mentioned the RX side of RLC gets PDUs and create status packet, and send it to sender (TX) since only the TX side can send back status packets to the peer. Additionally TX side cannot send any kind of packets unless there is a transmission opportunity.

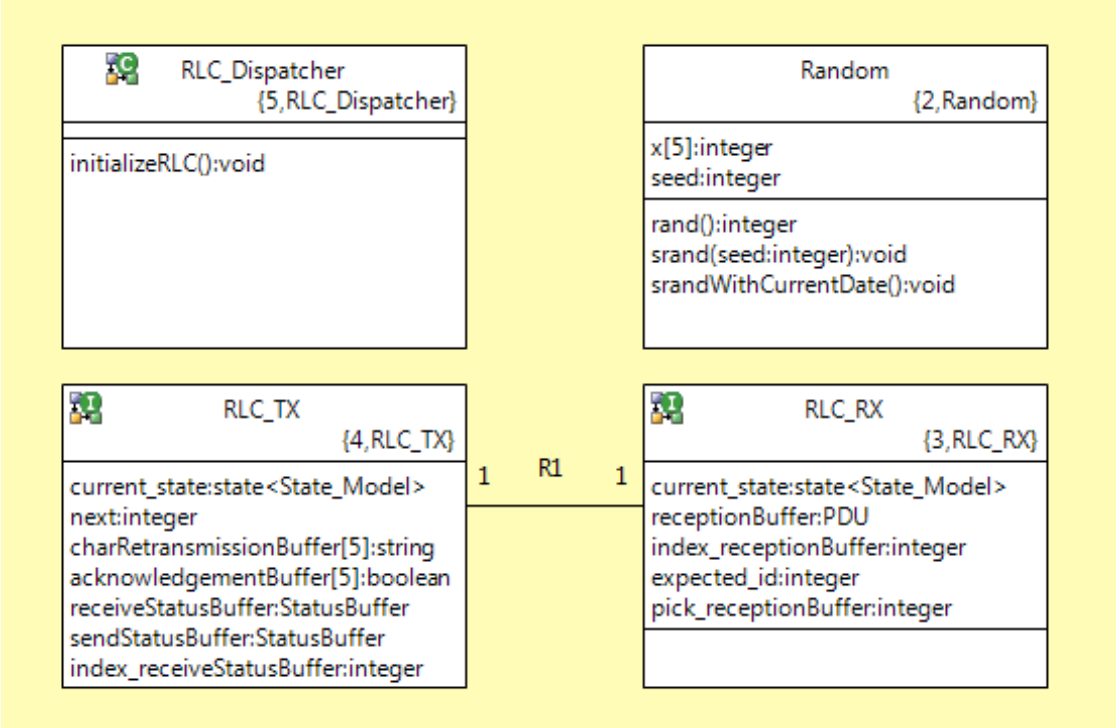


Figure 5.3.8.1 – RLC (AM) class diagram

A transmission opportunity is a signal from a MAC layer which specifies which entity is allowed to send packets. MAC layer sends transmission opportunity based on the network's traffic, bandwidth and size of packets (which is estimated by RLC). Thus in a real heterogeneous system, this given transmission opportunity might not exist after the packet is prepared for sending by RLC, since traffic in lower layer might be changed (for more info see Appendix B).

The *Receiver_TestCase* simulated the transmission opportunity in the *Generate* state, by randomly sending *tx_OP* signals through the MAC interface. In addition *receivePDU(id, char)* signals with random id and char parameters are sent as before.

As shown in figure 5.3.8.2 the RX side of RLC starts from the *Idle* and as soon as a *receivePDU(id, char)* signal has been received packets are buffered in *ReceptionBuffer* state. Later RLC checks the buffered packet and if it was the expected one, the packet would be sent to upper layer through PDCP interface. In case of other packets existed in buffer which are supposed to be sent, RLC sends them as well. Eventually status packets must be created. RLC decides about creating ACK or NACK in *SendToPDCP*. When expected packets existed in reception buffer RLC sends them one by one via *receiveSDU(char)* signal and create an ACK status with id as a next expected packet, otherwise a NACK status will be created for the current expected packet. The created status packets are buffered in *sendStatusBuffer* which is a buffer in TX side of RLC and remain there until the TX side gets a transmission opportunity. RLC ends up in the *Idle* state again after it passed through the *BufferSendStatus* state.

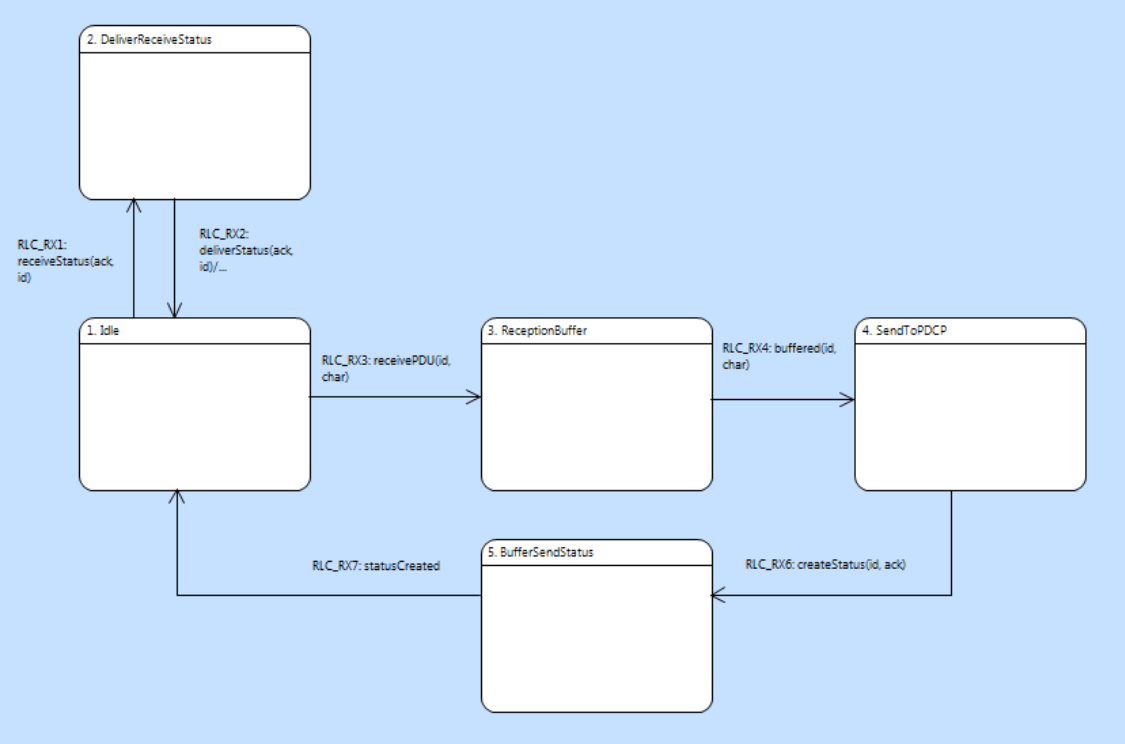


Figure 5.3.8.2 - RLC_RX instance based state machine

On the other hand if a *tx_OP* signal is raised by the test case, the TX side of RLC checks if there is any created status in *sendStatusBuffer* within the state

Check_SendStatusBuffer. In case of existence RLC sends the first created status which has not been sent before.

As for iteration five, *Receiver_TestCase* validates three different scenarios (states are illustrated in Figure 5.3.8.3):

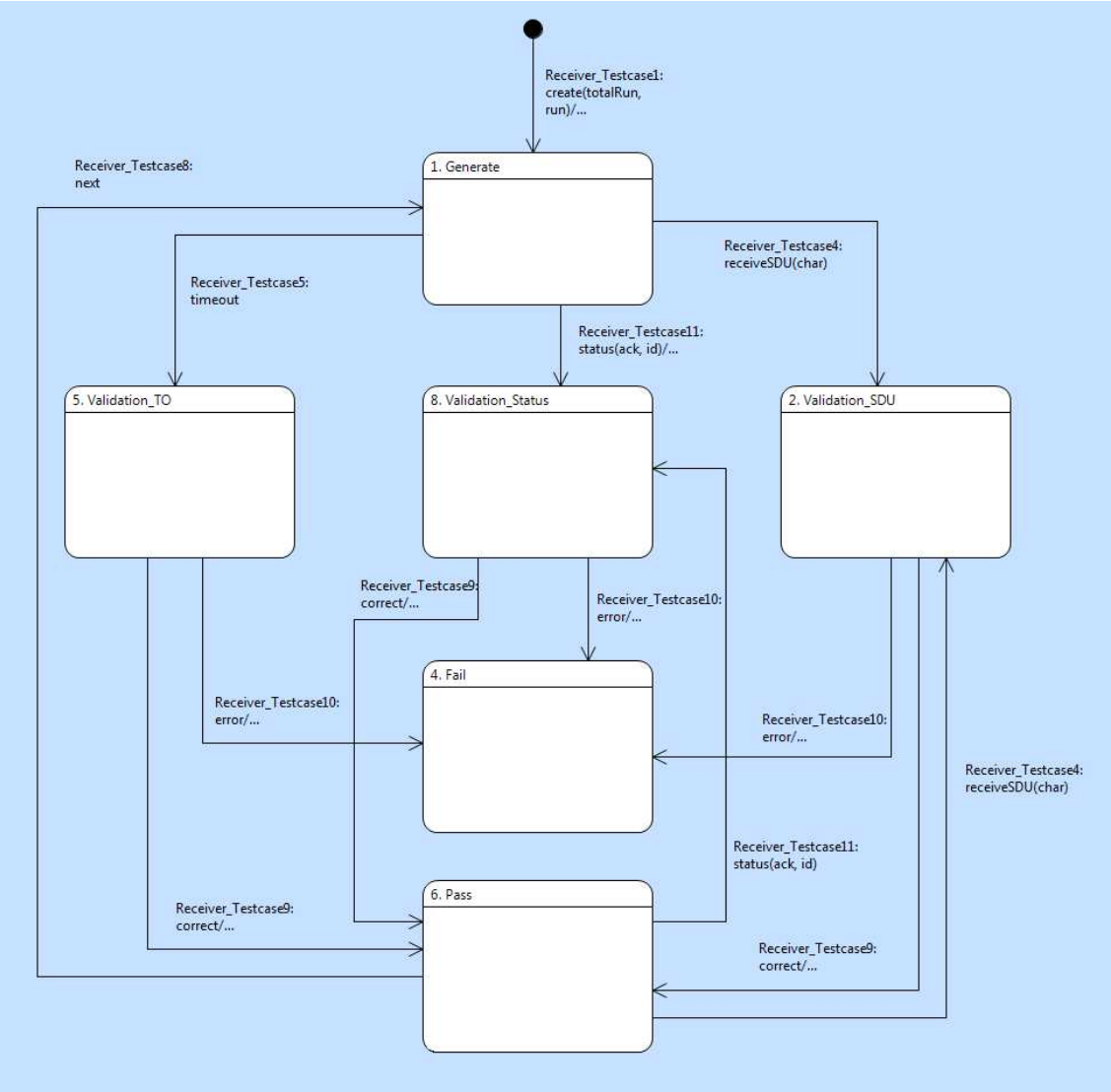


Figure 5.3.8.3 - Receiver_TestCase Instance based state machine

- Validation_SDU: When test case received a *receiveSDU(char)* signal, char parameter is compared with the expected letter and in case of equality test case ends up in *Pass* state. Contrarily difference of letters means that RLC has sent a wrong SDU packet which leads the test case to *Fail* state.
- Validation_Status: Beware that test case has an array called, *received_acknowledgement* that is used to keep track of received status for each letter. If a *status(ack, id)* signal received, test case first figures out what was the status type. If it was a NACK, then first off, test case checks the last packet id that was sent before. If the NACK was not in scope of sent packet ids, then test case ends up in *Fail* state. This means that RLC sent back inappropriate NACK

back to test case (e.g. if the last signal was *sendPDU(E, 1)* getting a *status(false, 4)* is a wrong behavior of system). On the other hand if the NACK was a proper one (e.g. *status(false, 0)*), an element of *received_acknowledgement* that corresponds to the parameter id (e.g. index zero) would be assigned to False. This action specifies that there were no positive acknowledgements (i.e. ACK) for that particular letter. If the status was an ACK just like before the first thing is checking the scope and if the ACK was a proper one, all elements in *received_acknowledgement* array would be set to True up to parameter. In both types of status packets after the *received_acknowledgement* array was maintained, the test case goes to *Pass* state.

- Validation_TO (time out): Whenever the timer expired, test case ends up in *Pass* state, either if all SDUs have been sent to RLC or no SDU has been sent, otherwise if an SDU has been sent, RLC must respond with a status or receiveSDU signals.

A test is considered complete only if all SDUs (i.e. letters defined in input strings) has been sent, all SDUs has been received back in a correct sequence and also ACKs for all letter has been sent by RLC. If not, test case loops and generates more signals until all the conditions have been fulfilled.

5.3.9. Iteration 9

Duration: 2 Days

Goal: Transmission opportunity (the transmitter role of RLC)

The last iteration focused on the RLC with a transmitter role; also as mentioned in the previous iteration an extra functionality named *Transmission opportunity* has been added.

The following shadow buffers are defined on the *Transmitter_TestCase* to keep track of RLC's functionality in a black box testing:

- shadow_receiveStatusBuffer: Array from structured data type named *StatusBuffer* with two attributes *ack* (boolean) and *id*
- shadow_retransmissionBuffer: Array from structured data type named *PDU* with two attributes, *id* and *char*
- shadow_sduBuffer: String array to SDU letters

Since the test case acts like a peer receiver RLC, three different signals might randomly sent via the *Generate* state to the RLC under test. The signals are as follows:

- sendSDU(char): The *char* parameter represents letters in sequence (e.g. H, E, L, L, O). For each SDU that has been sent, test case keeps a copy in *shadow_sduBuffer*.
- receiveStatus(ack, id): After a status packet has been sent through MAC interface, test case checks the *id* parameter to determine whether the status

packet was an appropriate one, and if it was, the status packet would be stored in *shadow_receiveStatusBuffer*. In case of NACK status (i.e. ack = False) the relevancy of the status determined by checking the exact element in *shadow_retransmissionBuffer* with an index equal to *id* parameter. If the mentioned element was not empty, then a correct NACK has been sent and must be buffered. In case of ACK the same approach is applied except that the element with an index *id* minus one would be checked (e.g. for ACK 4, test case checks *shadow_retransmissionBuffer[id - 1]*).

- tx_OP(): Transmission opportunity which is being sent through MAC interface.

The state diagram of the RLC TX is illustrated in Figure 5.3.9.1. When RLC receives *sendSDU(char)* signals from the test case (i.e. PDCP) the TX side buffers each packet (e.g. letters in sequence) in the *sduBuffer*. The buffering process is done in *SDUBuffering* state.

Due to the fact that the RX side of RLC is responsible for receiving packets from MAC interface, *receivePDU(char, id)* and *receiveStatus(ack, id)* signals are handled by the RX side. Therefore each status packet that has been sent by test case caught by RX side and then delivered to TX side to be buffered in *receiveStatusBuffer*. The process of buffering is done via *ReceiveStatusBuffering* state but before buffering all incoming status packets, TX side checks the *id* and only appropriate packets would be buffered.

tx_OP is the only signal that passes through MAC interface and targets the TX side (for more information see Appendix B). The TX side is just able to buffer packets until it gets a transmission opportunity. Notice that there is a priority for sending packet after each transmission opportunity. The priority of sending packets after a transmission opportunity and operations within each state are described below:

- Check_SendStatusBuffer: As explained in iteration 8, this state is responsible for checking the created status packets by RX side of RLC. Therefore created status packets have the highest priority among other packets and if there is any status packets found in the *sendStatusBuffer*, TX passes through *found* transition and sends packets through MAC interface back to test case. In this iteration the *sendStatusBuffer* is empty (because no PDU have been sent to the RLC thus no status created as well) and *notfound()* event would be raised.
- Check_ReceiveStatusBuffer: In case of no created status, the opportunity of transmission is given to packets that have to be re-send. Therefore TX side checks the *receiveStatusBuffer* in search of received status packets. If there was an ACK in *receiveStatusBuffer*, RLC should not send any packets back, instead the ACK indicates that all packets that has been sent before from RLC, are received by its peer correctly. Thus RLC removes all PDU packets from *retransmissionBuffer* up to the *id* parameter designated by status packet and hands in the opportunity to the next state. When a NACK was founded in buffer, RLC checks the *retransmissionBuffer* for the designated status id, and if it was not empty then RLC expends the opportunity and re-sends the particular

PDU back to test case within *Re-SendToMAC* state. On the other hand if no status existed in buffer, transmission opportunity would be consumed by next state. (Notice that only proper ACK or NACK packets have been buffered)

- Check_SDUBuffer: The last buffer that TX checks, is the *sduBuffer*. If a SDU packet founded in this buffer, RLC makes a PDU copy from the packet (i.e. form a PDU), and place it on the retransmission buffer within the state called *CopyToRetransmissionBuffer*. When the PDU packet is created, the corresponding SDU packet will be removed from the buffer. Finally in the *SendPDUToMAC* state PDU packet is sent back to the test case via *sendPDU(id, char)* signal.

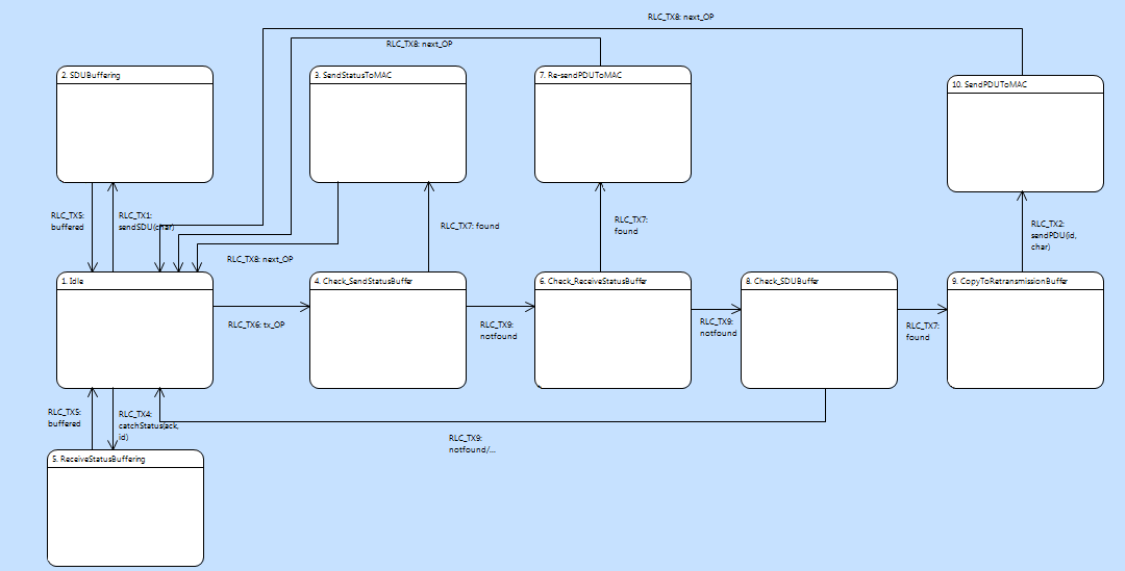


Figure 5.3.9.1 - RLC_TX instance based state machine

According to RLC's functionality, there are two validation phases in the test case (Figure 5.3.9.2). One for the *sendPDU(id, char)* signal and the other for time out.

Whenever the test case receives a *sendPDU(id, char)* signal it goes to *ValidationPDU* state. The first thing that the test case must consider is what has been sent to RLC. If no transmission opportunity was sent then, the test case ends up in the *Fail* state. That is because; RLC should not send any signal back without having a transmission opportunity. In case that the opportunity is given to RLC, test case must go through the priority of packets based on its shadow buffers. As mentioned the first priority was re-transmission of packets (Notice that the created status packets are omitted here due to the fact that RLC has a transmitter role). Thus test case searches in *shadow_receiveStatusBuffer*. If a NACK is found in buffer, test case checks the corresponding element in the *shadow_retransmissionBuffer* and if the element was empty, test case ends up in "Fail" state. Since there was nothing in the retransmission buffer but RLC has sent back a PDU packet. Otherwise the packet that was received by a *sendPDU(id, char)* signal is compared to the packet which supposed to be re-send and if they match the test will be passed.

Finally if there were no status packets in *shadow_receiveStatusBuffer*, test case checks the *shadow_sduBuffer* as a next priority level. In case there is a SDU packet in *shadow_sduBuffer* test case maintains the *shadow_retransmissionBuffer* (i.e. make a PDU copy in the shadow buffer). Then the PDU packet that has been received would be compared with the PDU packet in *shadow_retransmissionBuffer* and if they match then test case ends up in *Pass* state.

When there were no packets on all buffers then no PDU packet should be received at all, thus in this situation receiving a packet from RLC is a wrong behavior of the system and test case goes to the *Fail* state.

On the other hand, in the time out validation state, the test case ends up in the *Pass* state until an opportunity has been sent. Because RLC is not allowed to transmit any packet and might only buffer receiving packets therefore it always waits for an opportunity in the *Idle* state. As soon as a transmission opportunity has been sent to RLC, and the timer expired in the test case then all shadow buffers will be checked (considering the priority) to see if there was any packet that supposed to be sent/re-sent back to test case. Consequently the test case ends up in *Fail* state if a packet has been founded in buffers and goes to the *Pass* state contrarily.

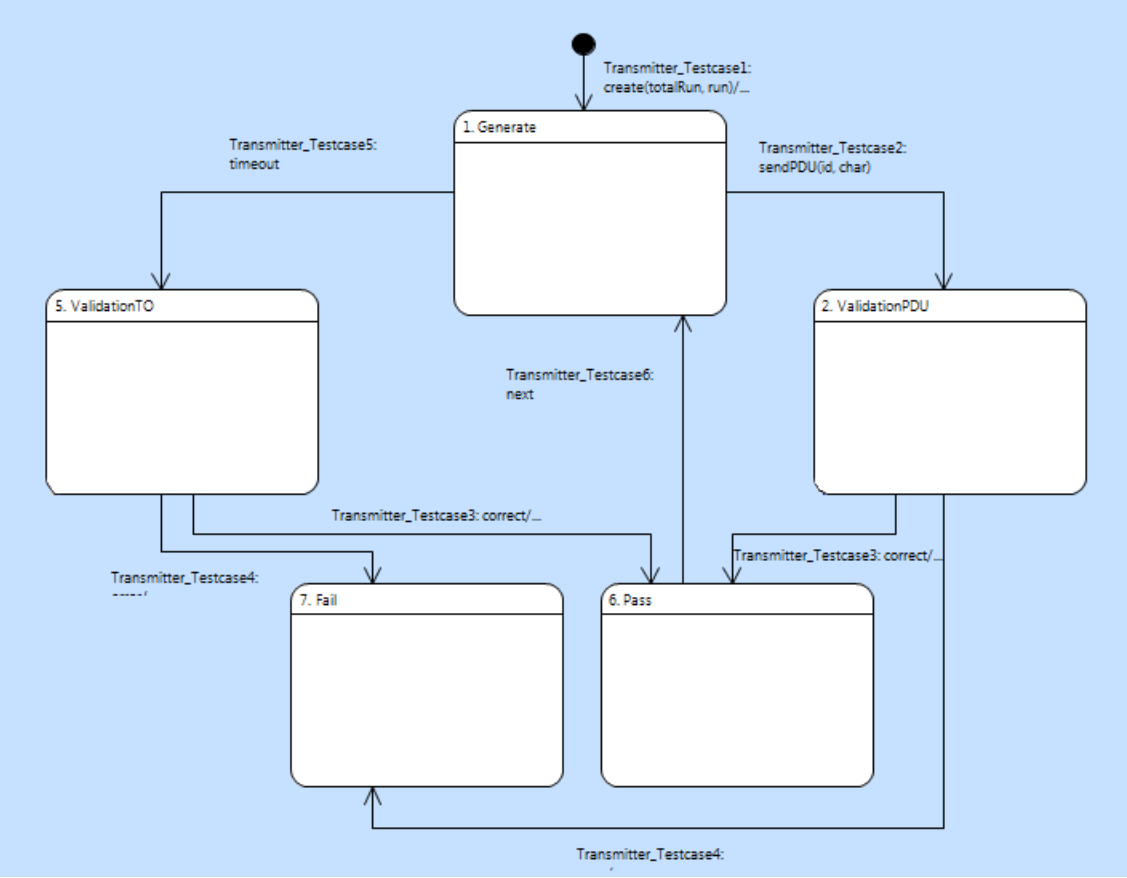


Figure 5.3.9.2 - Transmitter_TestCase instance based state machine

The test is completed when all SDU packets have been sent and all the corresponding PDU packets received by the test case. Bear in mind that only those PDU packets that

have got an acknowledgment signal are considered as received packets (i.e. their relevant flags in *received* array assigned to True).

6. Results

The suggested methodology was applied successfully on the modeling process of the RLC domain and reusability and reliability of the system has been validated by the case study.

The research question has been answered when the case study was accomplished. As an answer to the research question we can claim that although there are various methodologies to develop a complete PIM, the suggested methodology facilitates the creation of the platform independent models.

Furthermore we can claim that this methodology is applicable and also feasible to use in real world systems. Technically we have used and validated the methodology to produce a complete PIM for an RLC protocol which is a complex entity of existing telecommunication systems.

Moreover, we experienced that applying this methodology:

1. Reduced the difficulties to create a complete PIM. Difficulties that are forced by a complex system. This result is the main target of the suggested methodology.
2. Avoid designers to get stuck in the implementation details. The common agile iterations are focused on adding functionalities and test them in a rapid manner [10]. This may cause designers to concentrate more on detailed problems and consume too much time coping with specific problems rather than delivering a workable system.
3. Avoid missing deadlines and employ better time management. Going deep into details may cause lack of time and missing deadlines consequently.
4. Keeps the abstraction level as high as possible on consecutive iterations and the amount of added concrete functionalities as low as possible. Thus it makes the daily meetings short. Moreover, the necessity for the presence of different stakeholders with different fields of expertise in daily meetings has been reduced.
5. Makes the system reusable. Reusability is one of the greatest benefits when applying the suggested methodology. Unlike other agile processes which mainly focus on hacking new functionalities to the system, this methodology has more systematic vision for adding new features both to the test case and the target system. Therefore on new iterations many features which have been added can be reused (e.g. a general pattern for the test case).

7. Reflections

The first part of the research was dedicated to the common way for developing software systems in an iterative manner. Thus the process of modeling the case study was begun by defining the architecture of the system and after that we tried to add different functionalities as agile as possible in different iterations and test the updated system using a unit testing for different features.

Following this approach made us going deep into detailed features and therefore we focused more on the domain specific problems. Consequently for some requirements a huge amount of time has been consumed in order to develop those specific requirements without considering the testing and integration of various features within the system. This leads to producing an incomplete PIM that is poorly tested and might be error prone when translated to a PSM.

Another drawback that we often experienced was the fact that not only applying changes to the existing features but also adding new features to other parts of the system is quite hard; due to the late integration.

Furthermore for many domain specific issues we had to consult with the domain experts who might not be available all the time.

Several industrial experts commented that late integration of different components in the system or a product line is a major problem that companies face during the development of different systems. By applying the suggested methodology, the effect of this problem has been substantially reduced as shown in this case study.

8. Guidelines

This section mainly focuses on providing some guidelines based on the experiences that we gained during the research along with some common rules and design patterns within the software community. Although most of the aspects of this section have been covered within the software community but they are valuable for Ericsson and those who want to use or perform more research on this methodology.

8.1. Choosing the appropriate level of abstraction

Perhaps, the most important subject for applying the suggested method is choosing the right level of abstraction for consecutive iterations. As discussed before the main divergence of the suggested methodology from common agile processes is that, new features are added by considering an abstraction from details. Therefore we avoid detailed problems that are difficult to add in one go and cannot be fully tested during a certain iteration.

It is not clear how much details should be added in one iteration. Based on the number of teams and designers, their skills, the domain, competitive vendors and so forth the length and duration of an iteration might vary as well. But the crucial point is that,

according to the time plan and dedicated time frame for an iteration, features that take a long time to be implemented and cannot be handled within the specific time frame must not be added.

Another point that one must be aware of is that always one should try to reuse features that has been added on former iterations rather than implementing functionalities that must be changed a lot in upcoming iterations.

In our case study it seems feasible to choose concrete functionalities as little as possible for each iteration step.

8.2. Refactoring

As the system evolves and modified with new requirements, the created models of the system become more complex. This may cause lowering the quality of the design in terms of readability maintainability, extensibility, modularity, reusability [3] [20]. Refactoring technique is a remedy to improve quality in the mentioned aspects [20].
The main activities that designers must consider during a refactoring step are as follows [20]:

- Identify which parts of model should be refactored
- Guarantee that the applied refactoring preserves functional behaviors (consequently, refactored models should be tested with the same test cases as before)
- Apply the refactoring
- Maintain the consistency between the refactored model and the rest of system models

Note that the refactoring step has some effects on productivity, cost, and amount of effort that has been put on the development process [2].

In general as discussed within *iteration 5.3.7* refactoring can be applied as:

- Changing of names of attributes, variables, classes and so forth to make them more understandable.
- Removing unnecessary attributes, operations, variables, classes etc.
- Enhancing algorithms within action language.
- Introducing new design patterns (e.g. Defining a dispatcher class on iteration 7)
- Changing design decisions that has been made in order to simplify the requirements

8.3. Black box VS Gray box testing

According to [18] “...as the complexity and size of software grow, the time and effort required to do sufficient testing grow”. Thus, it is not feasible to apply manual testing on large systems. A remedy to this issue is automated and random testing.

As mentioned in the *Iterations* section, the technique that we used for testing the target system was aimed to be black box testing [19]. In our version all the possible inputs

including the proper and evil data will be randomly fed to the system through well-defined interfaces and expected or unexpected behavior of the system will be evaluated according to the outputs. The main advantages achieved by applying black box testing were:

- Simplicity: Testers do not need to understand the internal structure and algorithms of the target system.
- Clarity: Based on agreed interfaces and unambiguous use cases.
- Reducing the time for test case development: Testers are not concerned about identifying every possible path within the target system.

However, this case study was performed by a group of only two persons, and therefore the design of both the system and the test was done by the same group. This might have caused us to neglect some of the potential errors when modeling the test case. Therefore as discussed in *Iteration 5.3.6*, to gain more confidence on correctness of the system a mixture of black box testing and gray box testing [18] has been applied where data structures of the system were needed.

In general, we realized that black box technique which generates completely random inputs is feasible for testing large systems. The tool itself offers the possibility of performing white box testing [18] during the modeling process.

As a recommendation, test cases should be modeled by a team other than the same team modeling the system under test. In case that this is not possible gray box testing methods might be applied.

8.4. Tool assessment

Using BridgePoint as a tool for creating executable models has some limitations. These limitations forced us to reconsider some of the modeling designs and solutions when we realized our design into BridgePoint. The major expectations and features that designers want from BridgePoint (at the time of writing this report) are as follows:

- Ability to define Function Package for the entire system which is not limited to a specific component.
- Improve shortcomings while working with arrays.
- Improve shortcomings of timer instances
- Improve and extend built in data types
- Ability to instantiate multiple components from a single component (analogous to class instantiation)
- Make graphical lines (e.g. transitions, interfaces, relationships) straight keyboard shortcuts
- Remove the existing bugs such as, debugger tool when using timers, inability to undo formalization of components and interfaces after formalizing them, so that one can rename components.
- Ability to copy array elements into other arrays with different dimensions.

9. Conclusions

In this research, we followed MDA while modeling a large and complex system. Subsequently, the suggested methodology helped us to model the system in an iterative manner, so that at the end of each iteration an executable and fully tested PIM, was delivered. Additionally, we provided some guidelines on how to start the process of modeling along with different patterns such as the mentioned test pattern that not only can be reused on different iterations within this domain, but also used as a pattern in various product lines.

Generally, based on the analysis of the case study and results, it is feasible to apply the suggested methodology on complex systems in order to reduce the development complexity and cost along with improving the quality of the system.

Finally, we focused on providing an executable model independent of different deployment platforms (i.e. PIM) rather than other advantages of MDA. Nevertheless, in practice, model driven architecture and executable models are not mature enough currently, but we strongly believe, this might be the future of the software development that more research and effort needs to be dedicated to.

10. Future work

This thesis has not covered other modes of the RLC (TM and UM) along with other layers in telecommunication systems. It would be interesting to design and model other layers in order to test and integrate the whole system.

Investigation of different tools for creating executable UML models could be done later on to compare them with BridgePoint.

Also, this thesis has not covered the evaluation of the generated code, therefore another interesting area is the evaluation of the generated code and its performance compare to manual programming within the telecommunication domain.

Additionally, considering that this process of modeling was performed in a domain that had a well-defined specification, it would be challenging if the same methodology was applied in another domain with incomplete specifications, from the scratch.

References

[1] G. Booch, J. Rumbaugh, I. Jacobson,"The Unified Modeling Language User Guide", *Addison Wesley*, 2005

[2] M. Fowler, "Refactoring: Improving the Design of Existing Code", *Addison Wesley*, 1999

[3] S. W. Ambler, P. J. Sadalage, "The Process of Database Refactoring: Strategies for Improving Database Quality", *Addison Wesley*, 2006

[4] K. Beck, "Test Driven Development: By Example", *Addison Wesley*, 2002

[5] http://www.omg.org/mda/

[6] http://www.ibm.com/developerworks/rational/library/3100.html#notes

[7] S. J. Mellor, K. Scott, A. Uhl, D. Weise, "MDA Distilled: Principles of Model-Driven Architecture", *Addison Wesley*, 2004

[8] http://www.mentor.com/products/sm/model_development/bridgepoint/

[9] M. Fowler, K. Scott, "UML Distilled: A Brief Guide to the Standard Object Modeling Language", *Addison Wesley*, Volume 3, 2003

[10] M. Cohn, "Agile Estimating and Planning", *Prentice Hall PTR*, 2005

[11] L. Crispin; J. Gregory, "Agile Testing: A Practical Guide for Testers and Agile Teams*"*, *Addison Wesley*, 2008

[12] K. Beck, C. Andres, "Extreme Programming Explained: Embrace Change", *Addison Wesley*, 2008

[13] L. Williams, R. Kessler, "Pair Programming Illuminated", *Addison Wesley*, 2002

[14] J. Shore, S. Warden, "The Art of Agile Development", *O'Reilly Media*, 2007

[15] S. Mellor, M. Balcer, "Executable UML: A Foundation for Model-Driven Architecture", *Addison Wesley*, 2002

[16] G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", *Addison Wesley*, 2005

[17] Scott W. Amble, "Agile Modeling: A Brief Overview", *Workshop of the pUML-Group held together with the «UML»2001 on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, 2001

[18] Wang Linzhang Et al, "Generating Test Cases from UML Activity Diagram based on Gray-Box Method", *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, *IEEE*, 2009.

[19] British Computer Society, "Standard for Software Component Testin", *SIGIST*, 2001

[20] Mens, T., Tourwe, T., "A Survey of Software Refactoring", *IEEE Transactions on software engineering*, 2004

[21] http://agilemanifesto.org/

[22] http://www.3gpp.org/specifications

[23] Koen Claessan, John Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs", *ICFP '00 Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, Volume 35 Issue 9, Sept. 2000.

Appendix A

Object Action Language

Here are listings of all of the OAL activies in the model.

1. State Actions

1.1. RLC_Dispatcher

```
select any rlc_tx from instances of RLC_TX;
if(empty rlc_tx)
                    // Initialization
                    create object instance rlc_tx of RLC_TX;
                    create object instance rlc_rx of RLC_RX;
                    relate rlc_rx to rlc_tx across R1;

                    //Initialize TX Indices (0 means there is nothing in buffer yet)
                    rlc_tx.index_sendStatusBuffer = 0;
                    rlc_tx.index_receiveStatusBuffer = 0;
                    rlc_tx.index_sduBuffer = 0;
                    rlc_tx.pick_send_status = 0;
                    rlc_tx.pick_receive_status = 0;
                    rlc_tx.index_retransmissionBuffer = 0;
                    rlc_tx.pick_sdu = 0;

                    //Initialize sendStatusBuffer
                    i=0;
                    while(i < rlc_tx.sendStatusBuffer.id.length)
                                        rlc_tx.sendStatusBuffer.id[i] = 999;
                                        rlc_tx.sendStatusBuffer.ack[i] = false;
                                        i = i + 1;
                    end while;

                    //Initialize receiveStatusBuffer
                    i=0;
                    while(i < rlc_tx.receiveStatusBuffer.id.length)
                                        rlc_tx.receiveStatusBuffer.id[i] = 999;
                                        rlc_tx.receiveStatusBuffer.ack[i] = false;
                                        i = i + 1;
                    end while;

                    //Initialize sduBuffer
                    i=0;
                    while(i < rlc_tx.sduBuffer.length)
                                        rlc_tx.sduBuffer[i] = "%";
                                        i = i + 1;
                    end while;

                    //Initialize retransmissionBuffer
                    i=0;
                    while(i < rlc_tx.retransmissionBuffer.id.length)
                                        rlc_tx.retransmissionBuffer.id[i] = 999;
                                        rlc_tx.retransmissionBuffer.char[i] = "%";
                                        i = i + 1;
                    end while;
```

```
                    //Initialize RX Indices (0 means there is nothing in buffer yet)
                    rlc_rx.index_receptionBuffer = 0;
                    rlc_rx.pick_receptionBuffer = 0;
                    rlc_rx.expected_id = 0;

                    //Initialize receptionBuffer
                    i=0;
                    while(i < rlc_rx.receptionBuffer.id.length)
                                        rlc_rx.receptionBuffer.id[i] = 999;
                                        rlc_rx.receptionBuffer.char[i] = "%";
                                        i = i + 1;
                    end while;
                    generate RLC_TX1:sendSDU(char: param.char) to rlc_tx;
else
                    generate RLC_TX1:sendSDU(char: param.char) to rlc_tx;
end if;
```

1.2. RLC_Dispatcher

```
select any rlc_rx from instances of RLC_RX;
if(empty rlc_rx)
                    // Initialization
                    create object instance rlc_tx of RLC_TX;
                    create object instance rlc_rx of RLC_RX;
                    relate rlc_rx to rlc_tx across R1;

                    //Initialize TX Indices (0 means there is nothing in buffer yet)
                    rlc_tx.index_sendStatusBuffer = 0;
                    rlc_tx.index_receiveStatusBuffer = 0;
                    rlc_tx.index_sduBuffer = 0;
                    rlc_tx.pick_send_status = 0;
                    rlc_tx.pick_receive_status = 0;
                    rlc_tx.index_retransmissionBuffer = 0;
                    rlc_tx.pick_sdu = 0;

                    //Initialize sendStatusBuffer
                    i=0;
                    while(i < rlc_tx.sendStatusBuffer.id.length)
                                        rlc_tx.sendStatusBuffer.id[i] = 999;
                                        rlc_tx.sendStatusBuffer.ack[i] = false;
                                        i = i + 1;
                    end while;

                    //Initialize receiveStatusBuffer
                    i=0;
                    while(i < rlc_tx.receiveStatusBuffer.id.length)
                                        rlc_tx.receiveStatusBuffer.id[i] = 999;
                                        rlc_tx.receiveStatusBuffer.ack[i] = false;
                                        i = i + 1;
                    end while;

                    //Initialize sduBuffer
                    i=0;
                    while(i < rlc_tx.sduBuffer.length)
                                        rlc_tx.sduBuffer[i] = "%";
                                        i = i + 1;
                    end while;
```

```
                    //Initialize retransmissionBuffer
                    i=0;
                    while(i < rlc_tx.retransmissionBuffer.id.length)
                                        rlc_tx.retransmissionBuffer.id[i] = 999;
                                        rlc_tx.retransmissionBuffer.char[i] = "%";
                                        i = i + 1;
                    end while;

                    //Initialize RX Indices (0 means there is nothing in buffer yet)
                    rlc_rx.index_receptionBuffer = 0;
                    rlc_rx.pick_receptionBuffer = 0;
                    rlc_rx.expected_id = 0;

                    //Initialize receptionBuffer
                    i=0;
                    while(i < rlc_rx.receptionBuffer.id.length)
                                        rlc_rx.receptionBuffer.id[i] = 999;
                                        rlc_rx.receptionBuffer.char[i] = "%";
                                        i = i + 1;
                    end while;

                    generate RLC_RX3:receivePDU(id: param.id, char: param.char) to rlc_rx;
else
                    generate RLC_RX3:receivePDU(id: param.id, char: param.char) to rlc_rx;

end if;
```

1.3. RLC_Dispatcher

```
select any rlc_rx from instances of RLC_RX;
if(empty rlc_rx)
                    // Initialization
                    create object instance rlc_tx of RLC_TX;
                    create object instance rlc_rx of RLC_RX;
                    relate rlc_rx to rlc_tx across R1;

                    //Initialize TX Indices (0 means there is nothing in buffer yet)
                    rlc_tx.index_sendStatusBuffer = 0;
                    rlc_tx.index_receiveStatusBuffer = 0;
                    rlc_tx.index_sduBuffer = 0;
                    rlc_tx.pick_send_status = 0;
                    rlc_tx.pick_receive_status = 0;
                    rlc_tx.index_retransmissionBuffer = 0;
                    rlc_tx.pick_sdu = 0;

                    //Initialize sendStatusBuffer
                    i=0;
                    while(i < rlc_tx.sendStatusBuffer.id.length)
                                        rlc_tx.sendStatusBuffer.id[i] = 999;
                                        rlc_tx.sendStatusBuffer.ack[i] = false;
                                        i = i + 1;
                    end while;

                    //Initialize receiveStatusBuffer
                    i=0;
                    while(i < rlc_tx.receiveStatusBuffer.id.length)
                                        rlc_tx.receiveStatusBuffer.id[i] = 999;
                                        rlc_tx.receiveStatusBuffer.ack[i] = false;
```

```
                    i = i + 1;
        end while;

        //Initialize sduBuffer
        i=0;
        while(i < rlc_tx.sduBuffer.length)
                    rlc_tx.sduBuffer[i] = "%";
                    i = i + 1;
        end while;

        //Initialize retransmissionBuffer
        i=0;
        while(i < rlc_tx.retransmissionBuffer.id.length)
                    rlc_tx.retransmissionBuffer.id[i] = 999;
                    rlc_tx.retransmissionBuffer.char[i] = "%";
                    i = i + 1;
        end while;

        //Initialize RX Indices (0 means there is nothing in buffer yet)
        rlc_rx.index_receptionBuffer = 0;
        rlc_rx.pick_receptionBuffer = 0;
        rlc_rx.expected_id = 0;

        //Initialize receptionBuffer
        i=0;
        while(i < rlc_rx.receptionBuffer.id.length)
                    rlc_rx.receptionBuffer.id[i] = 999;
                    rlc_rx.receptionBuffer.char[i] = "%";
                    i = i + 1;
        end while;
        generate RLC_RX1:receiveStatus(ack: param.ack, id: param.id) to rlc_rx;
else
        generate RLC_RX1:receiveStatus(ack: param.ack, id: param.id) to rlc_rx;
end if;
```

1.4. RLC_Dispatcher

```
select any rlc_tx from instances of RLC_TX;
if(empty rlc_tx)
        // Initialization
        create object instance rlc_tx of RLC_TX;
        create object instance rlc_rx of RLC_RX;
        relate rlc_rx to rlc_tx across R1;

        //Initialize TX Indices (0 means there is nothing in buffer yet)
        rlc_tx.index_sendStatusBuffer = 0;
        rlc_tx.index_receiveStatusBuffer = 0;
        rlc_tx.index_sduBuffer = 0;
        rlc_tx.pick_send_status = 0;
        rlc_tx.pick_receive_status = 0;
        rlc_tx.index_retransmissionBuffer = 0;
        rlc_tx.pick_sdu = 0;

        //Initialize sendStatusBuffer
        i=0;
        while(i < rlc_tx.sendStatusBuffer.id.length)
                    rlc_tx.sendStatusBuffer.id[i] = 999;
                    rlc_tx.sendStatusBuffer.ack[i] = false;
```

```
                    i = i + 1;
        end while;

        //Initialize receiveStatusBuffer
        i=0;
        while(i < rlc_tx.receiveStatusBuffer.id.length)
                    rlc_tx.receiveStatusBuffer.id[i] = 999;
                    rlc_tx.receiveStatusBuffer.ack[i] = false;
                    i = i + 1;
        end while;

        //Initialize sduBuffer
        i=0;
        while(i < rlc_tx.sduBuffer.length)
                    rlc_tx.sduBuffer[i] = "%";
                    i = i + 1;
        end while;

        //Initialize retransmissionBuffer
        i=0;
        while(i < rlc_tx.retransmissionBuffer.id.length)
                    rlc_tx.retransmissionBuffer.id[i] = 999;
                    rlc_tx.retransmissionBuffer.char[i] = "%";
                    i = i + 1;
        end while;

        //Initialize RX Indices (0 means there is nothing in buffer yet)
        rlc_rx.index_receptionBuffer = 0;
        rlc_rx.pick_receptionBuffer = 0;
        rlc_rx.expected_id = 0;

        //Initialize receptionBuffer
        i=0;
        while(i < rlc_rx.receptionBuffer.id.length)
                    rlc_rx.receptionBuffer.id[i] = 999;
                    rlc_rx.receptionBuffer.char[i] = "%";
                    i = i + 1;
        end while;
        generate RLC_TX6:tx_OP() to rlc_tx;
else
        generate RLC_TX6:tx_OP() to rlc_tx;
end if;
```

1.5. RLC_RX State[2]:DeliverReceiveStatus

```
// Here RLC RX receives status signals from the MAC
//          and send it away RLC TX to transmitt it

generate RLC_RX2:deliverStatus(ack: param.ack, id: param.id) to self;
```

1.6. RLC_RX State[3]:ReceptionBuffer

```
//Check whether this PDU(id, char) have already existed in the buffer or not
i = 0;
flag = false;
while(i < self.index_receptionBuffer)
        if(self.receptionBuffer.id[i] == param.id)
                    flag = true;
```

```
                                        break;
                    end if;
                    i = i + 1;
end while;

if(flag == false)
                    //Buffer received PDUs
                    self.receptionBuffer.id[self.index_receptionBuffer] = param.id;
                    self.receptionBuffer.char[self.index_receptionBuffer] = param.char;
                    self.index_receptionBuffer = self.index_receptionBuffer + 1;
end if;
// go to create status
generate RLC_RX4:buffered(id: param.id, char: param.char) to self;
```

1.7. RLC_RX State[4]:SendToPDCP

```
// Check whether ACK or NACK have to be created
if(self.expected_id == param.id)
                    //Check if we have more PDUs that we should create ACK for them
                    i = 0;
                    while(i < self.index_receptionBuffer)
                                        if(self.receptionBuffer.id[i] == self.expected_id)
                                                            PDCP_RLC_Port::receiveSDU(char:
self.receptionBuffer.char[i]);
                                                            //LOG
                                                            LOG::LogInfo(message: "RLC send SDU to Port: " +
self.receptionBuffer.char[i]);
                                                            //LOG
                                                            self.expected_id = self.expected_id + 1;
                                                            i = -1;
                                        end if;
                                        i = i + 1;
                    end while;

                    generate RLC_RX6:createStatus(id: self.expected_id, ack: true) to self;
                    LOG::LogInfo(message: "Creating ACK");
                    LOG::LogInteger(message: self.expected_id);
else
                    generate RLC_RX6:createStatus(id: self.expected_id, ack: false) to self;
                    LOG::LogInfo(message: "Creating NACK");
                    LOG::LogInteger(message: self.expected_id);
end if;
```

1.8. RLC_RX State[5]:BufferSendStatus

```
//create ACK or NACK
select one tx related by self->RLC_TX[R1];
                    if(not empty tx)
                                        tx.sendStatusBuffer.id[tx.index_sendStatusBuffer] = param.id;
                                        tx.sendStatusBuffer.ack[tx.index_sendStatusBuffer] = param.ack;
                                        tx.index_sendStatusBuffer = tx.index_sendStatusBuffer + 1;

                    else
                                        LOG::LogInfo(message: "No object of RLC_TX exists");
end if;
// Back to idle
generate RLC_RX7:statusCreated() to self;
```

1.9. RLC_RX

```
//Delivering status to RLC_TX
select one tx related by self->RLC_TX[R1];
if(not empty tx)
                    generate RLC_TX4:catchStatus(ack: param.ack, id:param.id) to tx;
else
                    LOG::LogInfo(message: "No object of RLC_TX exists");
end if;
```

1.10. RLC_TX State[2]:SDUBuffering

```
// Buffer receiving SDUs
self.sduBuffer[self.index_sduBuffer] = param.char;
self.index_sduBuffer = self.index_sduBuffer + 1;

LOG::LogInfo(message: "RLC: Buffered " + param.char + " in sduBuffer");

// Back to idle
generate RLC_TX5:buffered() to self;
```

1.11. RLC_TX State[3]:SendStatusToMAC

```
// Send Status Signal to MAC and increment pick_status
if(self.pick_send_status < self.sendStatusBuffer.id.length)
                    send MAC_RLC_Port::status(id: self.sendStatusBuffer.id[self.pick_send_status],
                                        ack: self.sendStatusBuffer.ack[self.pick_send_status]);
                    // LOG
                    if(self.sendStatusBuffer.ack[self.pick_send_status] == true)
                                        LOG::LogInfo(message: "RLC: send status ACK");
                                        LOG::LogInteger(message: self.sendStatusBuffer.id[self.pick_send_status]);
                    else
                                        LOG::LogInfo(message: "RLC: send status NACK");
                                        LOG::LogInteger(message: self.sendStatusBuffer.id[self.pick_send_status]);
                    end if;
                    // LOG

                    self.pick_send_status = self.pick_send_status + 1;
else
                    LOG::LogInfo(message: "sendStatusBuffer is overflowed!");
end if;
// Go back to idle
generate RLC_TX8:next_OP() to self;
```

1.12. RLC_TX State[4]:Check_SendStatusBuffer

```
// Check if there is any status in sendStatusBuffer that must be sent
if((self.index_sendStatusBuffer > 0) and
                    (self.sendStatusBuffer.id[self.pick_send_status] != 999))
                    generate RLC_TX7:found() to self;
else
                    generate RLC_TX9:notfound() to self;
end if;
```

1.13. RLC_TX State[5]:ReceiveStatusBuffering

```
// Check if it is a crapy Status or not (avoid buffer overflow attacks)
if(param.ack == true)
```

```
                    if(param.id > 0)
                                        if(self.retransmissionBuffer.char[param.id - 1] == "%")
                                                            // Crapy Status discard it
                                                            LOG::LogInfo(message: "RLC: Discard crapy ACK");
                                                            generate RLC_TX5:buffered() to self;
                                        else
                                                            // Buffer Status that we get from the peer
                                                            self.receiveStatusBuffer.id[self.index_receiveStatusBuffer] =
param.id;
                                                            self.receiveStatusBuffer.ack[self.index_receiveStatusBuffer] =
param.ack;
                                                            self.index_receiveStatusBuffer =
self.index_receiveStatusBuffer + 1;
                                                            LOG::LogInfo(message: "RLC: Buffered ACK");
                                                            LOG::LogInteger(message: param.id);
                                                            // Back to idle
                                                            generate RLC_TX5:buffered() to self;
                                        end if;
                    else
                                        // Crapy Status discard it (ACK 0)
                                        LOG::LogInfo(message: "RLC: Discard crapy ACK");
                                        generate RLC_TX5:buffered() to self;
                    end if;
elif(param.ack == false)
                    if(self.retransmissionBuffer.char[param.id] == "%")
                                        // Crapy Status discard it
                                        LOG::LogInfo(message: "RLC: Discard crapy NACK");
                                        generate RLC_TX5:buffered() to self;
                    else
                                        // Buffer Status that we get from the peer
                                        self.receiveStatusBuffer.id[self.index_receiveStatusBuffer] = param.id;
                                        self.receiveStatusBuffer.ack[self.index_receiveStatusBuffer] = param.ack;
                                        self.index_receiveStatusBuffer = self.index_receiveStatusBuffer + 1;
                                        LOG::LogInfo(message: "RLC: Buffered NACK");
                                        LOG::LogInteger(message: param.id);
                                        // Back to idle
                                        generate RLC_TX5:buffered() to self;
                    end if;
end if;
```

1.14. RLC_TX State[6]:Check_ReceiveStatusBuffer

```
//Check the received status buffer to determine what should we re-send

// we have to check if there is any status in "receivedStatusBuffer" or not
// This cannot be simply checked by "index_receiveStatusBuffer > 0" because
// after first time that we received an status then "index_receiveStatusBuffer"
// won"t be zero again
tempAckID = self.receiveStatusBuffer.id[self.pick_receive_status];
tempAck = self.receiveStatusBuffer.ack[self.pick_receive_status];
// We check temp!=999 because if there was an empty Status on cell (pick_receive_status)
// then we get array out of bound in retransmissionBuffer.char[999]
if(tempAckID != 999)
                    // if we have NACK in buffer
                    if(tempAck == false)
                                        if(self.retransmissionBuffer.char[tempAckID] == "%")
                                                            generate RLC_TX9:notfound() to self;
                                        else
```

```
                                        generate RLC_TX7:found() to self;
                                end if;
                        elif(tempAck == true)
                                // If it is an ACK delete all PDUs in retransmisssion buffer until the ackID
                                if(self.retransmissionBuffer.char[tempAckID - 1] != "%")
                                        j = self.receiveStatusBuffer.id[self.pick_receive_status];
                                        i = 0;
                                        while(i < j)
                                                self.retransmissionBuffer.id[i] = 999;
                                                self.retransmissionBuffer.char[i] = "%";
                                                i = i + 1;
                                        end while;
                                        LOG::LogInfo(message: "RLC: maintaining
retransmissionBuffer for ACK");
                                        LOG::LogInteger(message: j);
                                        self.pick_receive_status = self.pick_receive_status + 1;
                                else
                                        LOG::LogInfo(message: "RLC: Wrong  ACK has been
buffered");
                                end if;
                                generate RLC_TX9:notfound() to self;
                        end if;
else
                        generate RLC_TX9:notfound() to self;
end if;
```

1.15. RLC_TX State[7]:Re-sendPDUToMAC

```
if(self.pick_receive_status < self.receiveStatusBuffer.id.length)
                        // If the picked status is a NACK find it in Retransmission Buffer and re-send it again
                        if(self.receiveStatusBuffer.ack[self.pick_receive_status] == false)
                                i = self.receiveStatusBuffer.id[self.pick_receive_status];
                                send MAC_RLC_Port::sendPDU(id: self.retransmissionBuffer.id[i],
                                        char: self.retransmissionBuffer.char[i]);
                                LOG::LogInfo(message: "RLC: re-send " + self.retransmissionBuffer.char[i]);
                                LOG::LogInteger(message: self.retransmissionBuffer.id[i]);
                                self.pick_receive_status = self.pick_receive_status + 1;
                        end if;
else
                        LOG::LogInfo(message: "receiveStatusBuffer is overflowed!");
end if;
// Back to idle
generate RLC_TX8:next_OP() to self;
```

1.16. RLC_TX State[8]:Check_SDUBuffer

```
// Check if sduBuffer is empty
if(self.index_sduBuffer > 0)
                        i = 0;
                        flag = false;
                        while(i < self.sduBuffer.length)
                                if(self.sduBuffer[i] != "%")
                                        flag = true;
                                        break;
                                end if;
                                i = i + 1;
                        end while;
                        if(flag == true)
```

```
                                        generate RLC_TX7:found() to self;
                    else
                                        generate RLC_TX9:notfound() to self;
                    end if;
else
                    generate RLC_TX9:notfound() to self;
end if;
```

1.17. RLC_TX State[9]:CopyToRetransmissionBuffer

```
// Copy the SDU that is ready to send in Re-Transmisssion Buffer and increment pick_sdu
self.retransmissionBuffer.char[self.index_retransmissionBuffer] = self.sduBuffer[self.pick_sdu];
self.retransmissionBuffer.id[self.index_retransmissionBuffer] = self.pick_sdu;
self.sduBuffer[self.pick_sdu] = "%";
self.pick_sdu = self.pick_sdu + 1;

// Go to SendPDUToMAC State and increment index_retransmissionBuffer
generate RLC_TX2:sendPDU(id: self.retransmissionBuffer.id[self.index_retransmissionBuffer],
                                                                        char:
self.retransmissionBuffer.char[self.index_retransmissionBuffer]) to self;
self.index_retransmissionBuffer = self.index_retransmissionBuffer + 1;
```

1.18. RLC_TX State[10]:SendPDUToMAC

```
// Send Signal to the Port
send MAC_RLC_Port::sendPDU(id:param.id , char: param.char);
LOG::LogInfo(message: "RLC: send PDU " + param.char);
LOG::LogInteger(message: param.id);
//Back to Idle and wait for next TX_OP
generate RLC_TX8:next_OP() to self;
```

1.19. RLC_TX

```
LOG::LogInfo(message: "RLC: There is nothing to send");
```

1.20. Receiver_Testcase State[1]:Generate

```
// 50 percent chance for sending TX_OP or sending PDU
create object instance r of Random;
r.srandWithCurrentDate();
self.randomtx_OP = r.rand()%100;
if(self.randomtx_OP <= 50)
                    //send TX_OP
                    MAC_RLC_Port::tx_OP();
                    LOG::LogInfo(message: "Testcase sent tx_OP");
elif(self.randomtx_OP > 50)
                    //send PDU
                    delete object instance r;
                    create object instance r of Random;
                    r.srandWithCurrentDate();
                    // Send PDU(id, char) which ids can be numbers between 0 to 4 randomly
                    self.randomPDUid = r.rand()%5;
                    send MAC_RLC_Port::receivePDU(id:self.randomPDUid, char:
self.inputString[self.randomPDUid]);
                    self.sent[self.randomPDUid]=true;// The flag for the sent character should set to True

                    //LOG
                    LOG::LogInfo(message: "TestCase sent: " + self.inputString[self.randomPDUid]);
```

```
                    LOG::LogInteger(message: self.randomPDUid);
                    //LOG
                    delete object instance r;
end if;

//set the timer
create event instance timeout of Receiver_Testcase5:timeout to self;
self.timer = TIM::timer_start(microseconds: 5000000, event_inst: timeout);
```

1.21. Receiver_Testcase State[2]:Validation_SDU

```
//Stop timer
st = TIM::timer_cancel(timer_inst_ref: self.timer);

//check if received character is the same as what has been sent before
if(self.inputString[self.next] == param.char)
                    self.received[self.next]=true;
                    self.next=self.next+1;
                    generate Receiver_Testcase9:correct() to self;
else
                    generate Receiver_Testcase10:error() to self;
end if;
```

1.22. Receiver_Testcase State[4]:Fail

```
self.run = self.run + 1;
LOG::LogInfo(message:"%%% Error occured --  wrong behaviour of system");
```

1..23. Receiver_Testcase State[5]:Validation_TO

```
//check if we received next(expected one) and we received Timeout as well then Error
if(self.next == self.sent.length) //Avoid array out of bound
                    generate Receiver_Testcase9:correct() to self;
else
                    if(self.sent[self.next] == true)
                                        generate Receiver_Testcase10:error() to self;
                    else
                                        generate Receiver_Testcase9:correct() to self;
                    end if;
end if;
```

1.24. Receiver_Testcase State[6]:Pass

```
self.run = self.run + 1;

//Decide whether test is completed
i=0;
flag1=true;
flag2=true;
flag3=true;
while(i<self.sent.length)
                    if (self.sent[i]==false)
                                        flag1=false;
                                        break;
                    end if;
                    i=i+1;
end while;
i = 0;
```

```
while(i<self.received.length)
        if (self.received[i]==false)
                flag2=false;
                break;
        end if;
        i=i+1;
end while;

i = 0;
while(i<self.received_acknowledgement.length)
        if (self.received_acknowledgement[i]==false)
                flag3=false;
                break;
        end if;
        i=i+1;
end while;


if ((flag1==true) and (flag2==true) and (flag3==true))
        LOG::LogInfo(message: "----------Test Completed---------");
else
        generate Receiver_Testcase8:next() to self;
end if;
```

1.25. Receiver_Testcase State[8]:Validation_Status

```
//Stop timer
st = TIM::timer_cancel(timer_inst_ref: self.timer);

if(self.ack == false)
        // we get NACK
        if(self.ackID <= self.next)
                generate Receiver_Testcase9:correct() to self;
                // Save the NACK status results on received_acknowledgement
                if(self.ackID == self.received_acknowledgement.length)
                        i = 0;
                        while(i < self.ackID)
                                self.received_acknowledgement[i] = true;
                                i = i + 1;
                        end while;
                else
                        self.received_acknowledgement[self.ackID] = false;
                end if;
        else
                generate Receiver_Testcase10:error() to self;
        end if;
elif(self.ack == true)
        if(self.ackID <= self.next)
                generate Receiver_Testcase9:correct() to self;
                // Save the ACK status results on received_acknowledgement
                i = 0;
                while(i < self.ackID)
                        self.received_acknowledgement[i] = true;
                        i = i + 1;
                end while;
        else
                generate Receiver_Testcase10:error() to self;
```

```
                    end if;
end if;
```

1.26. Receiver_Testcase

```
// Keep track of how many times this testcase should be run
self.run = param.run;
self.totalRun=param.totalRun;

// Initialize the input array
self.inputString[0] = "H";
self.inputString[1] = "E";
self.inputString[2] = "L";
self.inputString[3] = "L";
self.inputString[4] = "O";

// Initialize the Sent and Received arrays
i=0;
while(i <5)
                    self.sent[i] = false;
                    self.received[i] = false;
                    self.received_acknowledgement[i] = false;
                    i = i + 1;
end while;

self.next=0;
self.ack = true; self.ackID = -1;
```

1.27. Receiver_Testcase

```
LOG::LogInfo(message: "Error --- Time out");
```

1.28. Receiver_Testcase

```
LOG::LogInfo(message: "State Pass - Time out validation");
```

1.29. Receiver_Testcase

```
// save the parameters

self.ack = param.ack;
self.ackID = param.id;
```

1.30. Receiver_Testcase

```
LOG::LogInfo(message: "Error --- ReceiveSDU Validation");
```

1.31. Receiver_Testcase

```
LOG::LogInfo(message: "Error --- Status Validation");
```

1.32. Receiver_Testcase

```
LOG::LogInfo(message: "State Pass - Status validation");
```

1.33. Receiver_Testcase

```
LOG::LogInfo(message: "State Pass - receiveSDU validation");
```

1.34. Receiver_Testcase

```
select any rTest from instances of Receiver_Testcase;
if (empty rTest)
		LOG::LogInfo(message:"No Object of Receiver_Testcase exists");
else
		Generate Receiver_Testcase4:receiveSDU(char: param.char) to rTest;
end if;
```

1.35. Receiver_Testcase

```
select any rTest from instances of Receiver_Testcase;
if (empty rTest)
		LOG::LogInfo(message:"No Object of Receiver_Testcase exists");
else
		Generate Receiver_Testcase11:status(ack: param.ack, id: param.id) to rTest;
end if;
```

1.36. Transmitter_Testcase State[1]:Generate

```
create object instance r of Random;
r.srandWithCurrentDate();
self.randomChoice = r.rand()%100;
self.randomAck = r.rand()%2;
delete object instance r;
//Weird thing about the Random function
// after third time of usage it only produced even numbers

create object instance r of Random;
r.srandWithCurrentDate();
self.randomAckID = r.rand()%6;
delete object instance r;

// Randomly choose between sendSDU or status or tx_OP
if(self.randomChoice <= 40)
		if(self.next < self.inputString.length) // Avoid array out of bound
				// send characters in sequence
				nextCharacter = self.inputString[self.next];
				LOG::LogInfo(message: "Transmitter_Testcase: SDU(" + nextCharacter + ")");
				send PDCP_RLC_Port::sendSDU(char: self.inputString[self.next]);
				// maintain shadow_sdubuffer and sent buffer
				self.sent[self.next] = true;
				self.next = self.next + 1;
				self.shadow_sduBuffer[self.index_shadow_sduBuffer] = nextCharacter;
				self.index_shadow_sduBuffer = self.index_shadow_sduBuffer + 1;
		end if;
elif((self.randomChoice > 40) and (self.randomChoice < 60))
		// randomly send ACK or NACK with random IDs
		if(self.randomAck == 0)
				LOG::LogInfo(message: "Transmitter_Testcase: ACK");
				LOG::LogInteger(message: self.randomAckID);
				send MAC_RLC_Port::receiveStatus(ack: true, id: self.randomAckID);

				// Maintain shodow_receiveBuffer
				if(self.randomAckID > 0)
						// Buffer the ACK if it was proper
						if(self.shadow_retransmissionBuffer.char[self.randomAckID -
1] != "%")
```

```
                    self.shadow_receiveStatusBuffer.id[self.index_shadow_receiveStatusBuffer] = self.randomAckID;

                    self.shadow_receiveStatusBuffer.ack[self.index_shadow_receiveStatusBuffer] = true;
                                                                    self.index_shadow_receiveStatusBuffer =
self.index_shadow_receiveStatusBuffer + 1;
                                                        end if;
                                        end if;
                    elif(self.randomAck == 1)
                                        LOG::LogInfo(message: "Transmitter_Testcase: NACK");
                                        LOG::LogInteger(message: self.randomAckID);
                                        send MAC_RLC_Port::receiveStatus(ack: false, id: self.randomAckID);
                                        // Maintain shodow_receiveBuffer
                                        if(self.shadow_retransmissionBuffer.char[self.randomAckID] != "%")
                                                            // Avoid buffering crapy status

                    self.shadow_receiveStatusBuffer.id[self.index_shadow_receiveStatusBuffer] = self.randomAckID;

                    self.shadow_receiveStatusBuffer.ack[self.index_shadow_receiveStatusBuffer] = false;
                                                        self.index_shadow_receiveStatusBuffer =
self.index_shadow_receiveStatusBuffer + 1;
                                        end if;
                    end if;
elif(self.randomChoice >= 60)
                    // send tx_OP
                    LOG::LogInfo(message: "Transmitter_Testcase: send tx_OP");
                    send MAC_RLC_Port::tx_OP();
end if;

//Set timer
create event instance timeout of Transmitter_Testcase5:timeout to self;
self.timer = TIM::timer_start(microseconds: 7000000, event_inst: timeout);
```

1.37. Transmitter_Testcase State[2]:ValidationPDU

```
// Stop Timer
st = TIM::timer_cancel(timer_inst_ref: self.timer);

// Check what we have sent
if(self.randomChoice <= 40)
                    // SDU has been sent, and we get PDU without tx_OP
                    generate Transmitter_Testcase4:error() to self;
elif((self.randomChoice > 40) and (self.randomChoice < 60))
                    // Status has been sent, and we get PDU without tx_OP
                    generate Transmitter_Testcase4:error() to self;
elif(self.randomChoice >= 60)
                    // We have sent tx_op
                    // First check if we have anything to re-send based on shadow_receiveStatusBuffer
//                  (check shadow_receiveStatusBuffer and shadow_retransmisssionBuffer)

                    if(self.shadow_receiveStatusBuffer.id[self.pick_shadow_receiveStatusBuffer] != 999)
                                        // There is something in shadow_status_buffer, Now check retransmissionBuffer
                                        tempAck =
self.shadow_receiveStatusBuffer.ack[self.pick_shadow_receiveStatusBuffer];
                                        tempAckID =
self.shadow_receiveStatusBuffer.id[self.pick_shadow_receiveStatusBuffer];
                                        if(tempAck == false)
                                                            // Picked status was a NACK, increment the
```

```
pick_shadow_receiveStatusBuffer
                                        self.pick_shadow_receiveStatusBuffer =
self.pick_shadow_receiveStatusBuffer + 1;
                                        if(self.shadow_retransmissionBuffer.char[tempAckID] !=
"%")
                                                // There was something to be re-sent
                                                if(param.id ==
self.shadow_retransmissionBuffer.id[tempAckID])
                                                        // Check if PDU we get
(param.id) is equal to what we had to re-send
                                                        self.received[param.id] =
true;
                                                        generate
Transmitter_Testcase3:correct() to self;
                                                else
                                                        generate
Transmitter_Testcase4:error() to self;
                                                end if;
                                        else
                                                // Nothing in retransmissionBuffer (No PDU
should be sent)
                                                generate Transmitter_Testcase4:error() to
self;
                                        end if;
                                end if;
                        else
                                //There is no PDU to be re-sent, Check SDU Buffer
                                if(self.index_shadow_sduBuffer > 0)
                                        // Maintain shadow_retransmissionBuffer

                        self.shadow_retransmissionBuffer.char[self.index_shadow_retransmissionBuffer] =
self.shadow_sduBuffer[self.pick_shadow_sduBuffer];

                        self.shadow_retransmissionBuffer.id[self.index_shadow_retransmissionBuffer] =
self.pick_shadow_sduBuffer;
                                        self.pick_shadow_sduBuffer = self.pick_shadow_sduBuffer +
1;
                                        self.index_shadow_retransmissionBuffer =
self.index_shadow_retransmissionBuffer + 1;
                                        //Check if the PDU we got(param.id) is equal to
//                                      the PDU that we have to send for the first time

                                        if(param.id ==
self.shadow_retransmissionBuffer.id[self.index_shadow_retransmissionBuffer - 1])
                                                //Maintain shadow_sduBuffer and received
buffer

                        self.shadow_sduBuffer[self.pick_shadow_sduBuffer - 1] = "%";
                                                generate Transmitter_Testcase3:correct() to
self;
                                        else

                                                generate Transmitter_Testcase4:error() to
self;
                                        end if;
                                else
                                        // There was nothing to send at all but we got PDU
                                        generate Transmitter_Testcase4:error() to self;
```

```
                                        end if;
                        end if;
end if;
```

1.38. Transmitter_Testcase State[5]:ValidationTO

```
// Check what has been sent before (tx_OP or SDU or ACK or NACK)

if(self.randomChoice <= 40)
                        // We have sent SDU, RLC buffered it and we get timeout
                        generate Transmitter_Testcase3:correct() to self;
elif((self.randomChoice > 40) and (self.randomChoice < 60))
                        // We have sent Status (ACK or NACK), RLC buffered it and we get timeout
                        generate Transmitter_Testcase3:correct() to self;
elif(self.randomChoice >= 60)
                        // We have sent tx_OP and got timeout

                        // Check if RLC had any received status to re-send PDU(search shadow_receiveStatusBuffer)
                        haveRetransmission = false;
                        haveSDU = false;
                        tempAck = self.shadow_receiveStatusBuffer.ack[self.pick_shadow_receiveStatusBuffer];
                        tempAckID = self.shadow_receiveStatusBuffer.id[self.pick_shadow_receiveStatusBuffer];
                        if(tempAckID != 999)
                                        // There is something on receiveStatusBuffer
                                        if(tempAck == true)
                                                        // We picked an ACK, nothing should be re-send
                                                        // Maintain received Buffer and retransmissionBuffer
                                                        i = 0;
                                                        while(i < tempAckID)
                                                                        self.shadow_retransmissionBuffer.id[i] =
999;
                                                                        self.shadow_retransmissionBuffer.char[i] =
"%";
                                                                        self.received[i] = true;
                                                                        i = i + 1;
                                                        end while;
                                                        self.pick_shadow_receiveStatusBuffer =
self.pick_shadow_receiveStatusBuffer + 1;
                                        else
                                                        // We picked a NACK
                                                        if(self.shadow_retransmissionBuffer.char[tempAckID] !=
"%")
                                                                        // There is a PDU on retransmissionBuffer
that must be re-sent for that NACK
                                                                        haveRetransmission = true;
                                                                        generate Transmitter_Testcase4:error() to
self;
                                                        else
                                                                        generate Transmitter_Testcase3:correct() to
self;
                                                        end if;
                                        end if;

                        // Check if RLC had any SDU to send (search shadow_sduBuffer)
                        elif(self.index_shadow_sduBuffer > 0)
                                        i=0;
                                        while(i < self.shadow_sduBuffer.length)
                                                        if(self.shadow_sduBuffer[i] != "%")
```

```
                                                haveSDU = true;
                                                break;
                                        end if;
                                        i = i + 1;
                                end while;
                                if(haveSDU == true)
                                        generate Transmitter_Testcase4:error() to self;
                                end if;
                        end if;
                        if((haveRetransmission == false) and (haveSDU == false))
                                generate Transmitter_Testcase3:correct() to self;
                        end if;
end if;
```

1.39. Transmitter_Testcase State[6]:Pass

```
//Decide whether test is completed
i=0;
flag1=true;
flag2=true;
while(i<self.sent.length)
                if (self.sent[i]==false)
                        flag1=false;
                        break;
                end if;
                i=i+1;
end while;
i = 0;
while(i<self.received.length)
                if(self.received[i]==false)
                        flag2=false;
                        break;
                end if;
                i=i+1;
end while;

if ((flag1==true) and (flag2==true))
                LOG::LogInfo(message: "----------Test Completed---------");
else
                generate Transmitter_Testcase6:next() to self;
end if;
```

1.40. Transmitter_Testcase State[7]:Fail

```
LOG::LogInfo(message:"%%% Error occured --  wrong behaviour of system");
self.run = self.run + 1;
```

1.41. Transmitter_Testcase

```
// Keep track of how many times this testcase should be run
self.run = param.run;
self.totalRun = param.totalRun;

// Initialize the inputString array
self.inputString[0] = "H";
self.inputString[1] = "E";
self.inputString[2] = "L";
self.inputString[3] = "L";
```

```
self.inputString[4] = "O";

// Initialize the shadow_receiveStatusBuffer, shadow_retransmissionBuffer
// and shadow_sduBuffer arrays

i = 0;
while(i < self.shadow_receiveStatusBuffer.id.length)
                    self.shadow_receiveStatusBuffer.id[i] = 999;
                    self.shadow_receiveStatusBuffer.ack[i] = false;
                    self.shadow_retransmissionBuffer.id[i] = 999;
                    self.shadow_retransmissionBuffer.char[i] = "%";
                    self.shadow_sduBuffer[i] = "%";
                    i = i + 1;
end while;

// Initialize the sent arrays
i = 0;
while(i < self.sent.length)
                    self.sent[i] = false;
                    self.received[i] = false;
                    i = i + 1;
end while;

// inputString array index (points to next character to send)
self.next = 0;

// Shadow Buffers indeces (0 means buffers are empty)
self.index_shadow_sduBuffer = 0;
self.index_shadow_receiveStatusBuffer = 0;
self.index_shadow_retransmissionBuffer = 0;
self.pick_shadow_sduBuffer = 0;
self.pick_shadow_receiveStatusBuffer = 0;
```

1.42. Transmitter_Testcase

```
LOG::LogInfo(message: "Time-out validation Error");
```

1.43. Transmitter_Testcase

```
LOG::LogInfo(message: "Testcase: State pass (TO validation)");
```

1.44. Transmitter_Testcase

```
LOG::LogInfo(message: "Testcase: State pass (PDU validation)");
```

1.45. Transmitter_Testcase

```
LOG::LogInfo(message: "PDU validation Error");
```

1.46. Transmitter_Testcase

```
select any tTest from instances of Transmitter_Testcase;
if(empty tTest)
                    LOG::LogInfo(message: "There is no object of Transmitter_Testcase exists");
else
                    generate Transmitter_Testcase2:sendPDU(id: param.id, char: param.char) to tTest;
end if;
```

2. Class Operations

2.1. RLC:AM:RLC_Dispatcher: initializeRLC

```
create object instance rlc_tx of RLC_TX;
create object instance rlc_rx of RLC_RX;
relate rlc_rx to rlc_tx across R1;

//Initialize TX Indices (0 means there is nothing in buffer yet)
rlc_tx.index_sendStatusBuffer = 0;
rlc_tx.index_receiveStatusBuffer = 0;
rlc_tx.index_sduBuffer = 0;
rlc_tx.pick_send_status = 0;
rlc_tx.pick_receive_status = 0;
rlc_tx.index_retransmissionBuffer = 0;
rlc_tx.pick_sdu = 0;

//Initialize sendStatusBuffer
i=0;
while(i < rlc_tx.sendStatusBuffer.id.length)
                        rlc_tx.sendStatusBuffer.id[i] = 0;
                        rlc_tx.sendStatusBuffer.ack[i] = false;
                        i = i + 1;
end while;

//Initialize receiveStatusBuffer
i=0;
while(i < rlc_tx.receiveStatusBuffer.id.length)
                        rlc_tx.receiveStatusBuffer.id[i] = 0;
                        rlc_tx.receiveStatusBuffer.ack[i] = false;
                        i = i + 1;
end while;

//Initialize sduBuffer
i=0;
while(i < rlc_tx.sduBuffer.length)
                        rlc_tx.sduBuffer[i] = "%";
                        i = i + 1;
end while;

//Initialize retransmissionBuffer
i=0;
while(i < rlc_tx.retransmissionBuffer.id.length)
                        rlc_tx.retransmissionBuffer.id[i] = 0;
                        rlc_tx.retransmissionBuffer.char[i] = "%";
                        i = i + 1;
end while;

//Initialize RX Indices (0 means there is nothing in buffer yet)
rlc_rx.index_receptionBuffer = 0;
rlc_rx.expected_id = 0;

//Initialize receptionBuffer
i=0;
while(i < rlc_rx.receptionBuffer.id.length)
                        rlc_rx.receptionBuffer.id[i] = 0;
                        rlc_rx.receptionBuffer.char[i] = "%";
```

```
                    i = i + 1;
end while;
```

2.2. TestFramework:TestCases:Random: rand

```
//*
//  * C style rand() function.
//  * Outputs a random unsigned integer.
//  * Idea from: http://www.agner.org/random/
//  * --> "Uniform random number generators in C++"
//  *
//  * This is a multiply-with-carry type of random number generator
//  * invented by George Marsaglia.
//  *
//  * It sort of works the same way. Although OAL
//  * limits certain things, like casting between different
//  * bit sizes. Although this implementation seems to work
//  * good enough for this little Hotel Project :)
//

sum =               2111111111 * self.x[3] +
                                    1492 * self.x[2] +
                                    1776 * self.x[1] +
                                    5115 * self.x[0] +
                                        self.x[4];

self.x[3] = self.x[2];
self.x[2] = self.x[1];
self.x[1] = self.x[0];
//self.x[4] = sum * ::pow(a: 2, n: 32);
self.x[4] = sum * (1073741824); // * 2^30, 2^32 overflows, this is the carry anyway
self.x[0] = sum;                                                    // Lower 32 bits of the sum

if(self.x[0] < 0)
                    return (-self.x[0]);
else
                    return self.x[0];
end if;
```

2.3. TestFramework:TestCases:Random: srand

```
//*
//  * srand(seed)
//  * Initiates the random function with
//  * the specified seed.
//  * Idea from: http://www.agner.org/random/
//  * --> "Uniform random number generators in C++"
//

i = 0;
s = param.seed;

// Make random numbers and put them into the buffers
while(i<5)
                    s = s * 28843829 - 1;
                    self.x[i] = s;
                    i = i+1;
end while;
```

```
// Randomize some more
i = 0;
while(i<19)
                    n = self.rand();
                    i = i+1;
end while;
```

2.4. TestFramework:TestCases:Random: srandWithCurrentDate

```
//
//   Seeds the random generator with the current date
//

d = TIM::current_date();
t = TIM::get_year(date: d) +
                    TIM::get_month(date: d) +
                    TIM::get_day(date: d) +
                    TIM::get_hour(date: d) +
                    TIM::get_minute(date: d) +
                    TIM::get_second(date: d);

self.srand(seed: t);
```

2.5. TestFramework:TestCases:Tracker: set_nr_Receiver_TestCase

```
self.nr_Receiver_TestCase=param.run;
```

2.6. TestFramework:TestCases:Tracker: get_nr_Transmitter_TestCase

```
return self.nr_Transmitter_TestCase;
```

2.7. TestFramework:TestCases:Tracker: get_nr_Receiver_TestCase

```
return self.nr_Receiver_TestCase;
```

2.8. TestFramework:TestCases:Tracker: setup

```
//setting up the system
// Initializing the number of running testcases
self.nr_Transmitter_TestCase = 0;
self.nr_Receiver_TestCase = 0;
//SS-Comented because just work on Receiver side
//Run testcases randomly
//create object instance r of Random;
//r.srandWithCurrentDate();

//temp=r.rand()%2;
//if (temp==0)
                    Generate Transmitter_Testcase1:create(totalRun: self.totalRun, run: self.nr_Transmitter_TestCase)
to Transmitter_Testcase creator;
//elif(temp==1)
                    //Generate Receiver_Testcase1:create(totalRun: self.totalRun, run: self.nr_Receiver_TestCase) to
Receiver_Testcase creator;
//end if;

LOG::LogInfo(message: "Testcases are instantiated");
```

2.9. TestFramework:TestCases:Tracker: set_runs

```
self.runs[param.run] = param.result;
```

2.10. TestFramework:TestCases:Tracker: get_runs

```
return self.runs[param.run];
```

3. Domain Functions

3.1. TestFramework: init

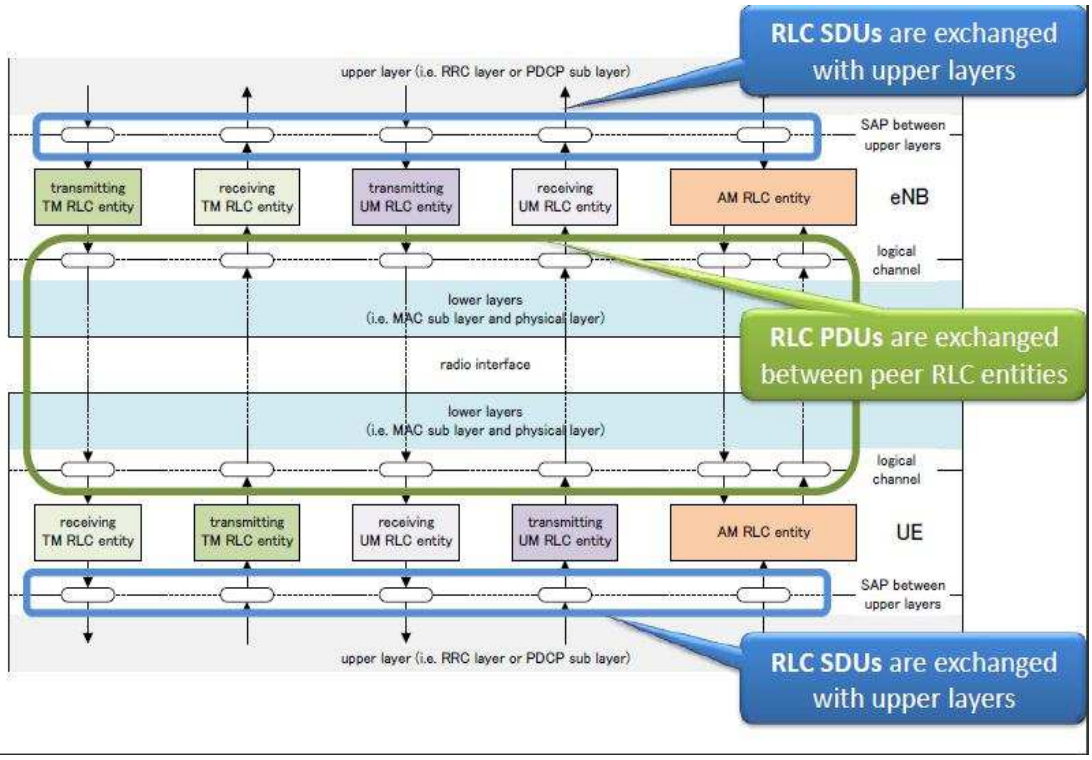
```
create object instance t of Tracker;
t.setup();
```

Appendix B

3GPP (3rd Generation Partnership Project) Radio Link Control Sub Layer:

RLC (Radio Link Control) Architecture:

An RLC entity receives RLC SDUs from upper layer and sends them to lower layer, and vice versa. The RLC could be a Control PDU or Data PDU. Data PDU can either be AMD PDU or AMD PDU segment, while Control PDU represents STATUS PDU. RLC communicates with upper layer (PDCP) through single SAP and after forming PDU, it sends it to lower layer (MAC). RLC entity communicates with its peer RLC entity via logical channel. RLC entity could be configured to work in three different modes: TM, UM and AM. This research will focus on AM RLC mode.



AM (Acknowledge Mode) RLC entity:

How AM RLC works:

An AM RLC consists of transmitting and receiving side. Transmitting side receives SDUs from upper layer and, after forming PDUs, sends them to its peer AM RLC via lower layer. Receiving side receives PDUs from its peer RLC via lower layer. After reassembling SDUs, it delivers them to upper layer.

These points apply to RLC entities:

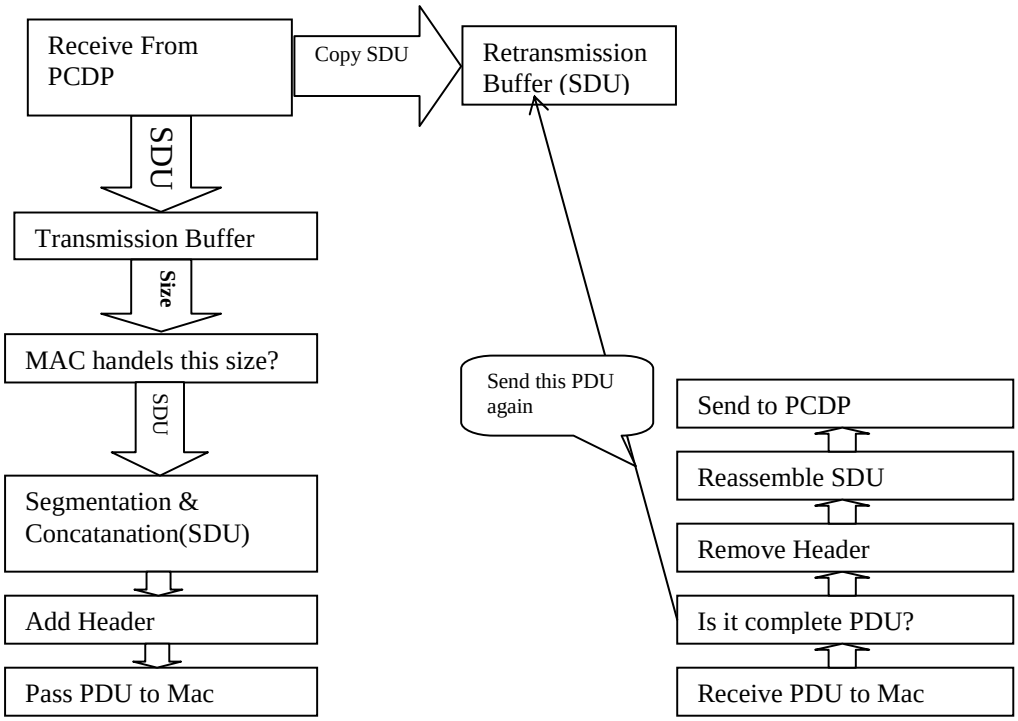
1. RLC SDUs are supposed to support byte aligned variable sizes.
2. RLC PDUs are formed only when notified by transmission opportunity from lower layer.

An AM RLC can be configured to communicate via two logical channels: DL/UL DCCH or DL/UL DTCH)

AM mode is suitable because it is:

1. Reliable for sequence delivery service
2. Suitable for carrying TCP traffic

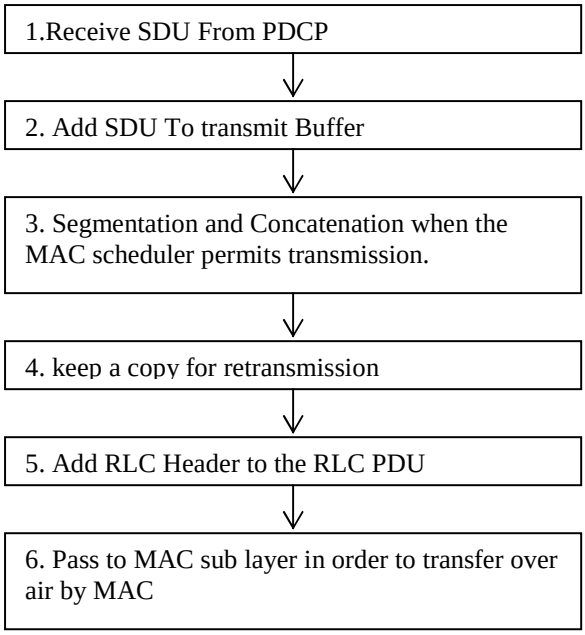
AM RLC is illustrated in the following chart:



The transmitting side supports transmission of RLC PDUs using ARQ(Automatic Repeat Request). It means that, when Transmitting side creates PDU for each opportunity, it should be re-segmented by RLC in case PDU doesn't fit notified size. The number of re-segmentation is not limited. In both cases, PDU formed from SDU or PDU segment formed from PDU should contain relevant header.

Receiving side, as soon as PDU has been received, checks whether it is received as duplication, and then reorders RLC PDUs in case they are received out of sequence, detect the loss of RLC PDUs at lower layer, requests transmission to its peer AM RLC entity, reassembles RLC SDUs from reordered RLC data PDUs and delivers SDUs to upper layer following the Sequence Number order.

An Overview on Data transmission in transmitting side of AM RLC:



Services which exist in this domain:

RLC provides a service for upper layer called data transfer. Lower layer provides two services: data transfer and notification, in order to announce transmission opportunity.

Functions which are supported by RLC layer:

- Transfer of upper layer PDUs
- Error correction through ARQ
- Concatenation, segmentation and reassembly of RLC SDUs
- Re-segmentation of RLC data PDUs
- Reordering of RLC data PDUs
- Duplicate detection
- RLC SDU discard
- RLC re-establishment
- Protocol error detection and recovery

The amount of data available for transmission:

First, MAC requests from RLC information about available data for transmission, in order to estimate size of PDU. Then, RLC calculates size according to these buffered data:

1. SDUs which are not yet formed as PDUs.
2. PDUs and PDU segments which are pending for retransmission.
3. In case of STATUS PDU is triggered but t-StatusProhibit is not running or has expired, RLC should estimate the size of STATUS PDU in next transmission opportunity.

Finally, MAC estimates PDU size according to results of RLC size calculation, and lower layer traffic.

At re-establishment time, receiving side should follow the following steps:

1. Reassemble RLC SDUs from the RLC data PDUs, which are received out of sequence, and deliver them to upper layer.
2. Discard any remaining data PDU which is not useful for reassembling.
3. Initialize relevant state variables and stop relevant timers.

Abbreviations:

VT (A): Acknowledged State Variable
VT (MS): Maximum Send State Variable
VT(S): Send State Variable
VR(R): Receive State Variable
VR (MR): Maximum Accepted Receive State Variable
VR(X): Reordering State Variable
VR (MS): Maximum STATUS Transmit State Variable
VR (H): Highest Received State Variable

AM Data transfer: (Transmit operations)

Transmitting side:

A) The transmitting side should prioritize data for transmission according to this:

1. Control Data PDU
2. Data PDU pending for Retransmission
3. Data PDU

B) The transmitting side should maintain window for transmission:

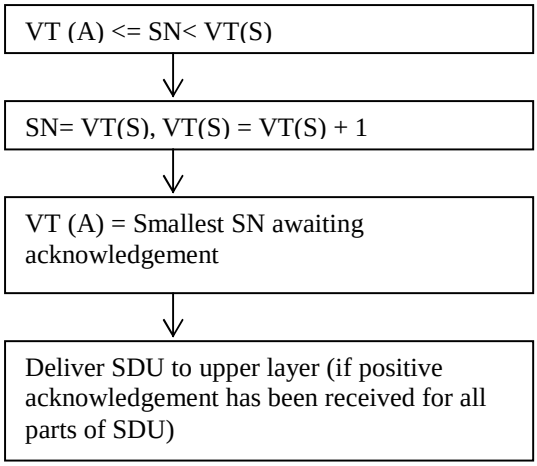
If VT (A) <= SN< VT(S), SN falls in transmission windows. Otherwise it falls outside.

Transmitting side should not send any PDUs with SN that falls outside of this window.

C) After delivery, transmitting side should set SN to VT(S) and then increment VT(S).

D) If PDU is successfully received by its peer RLC, this transmitter RLC will receive a positive acknowledgement as STATUS PDU from its peer RLC entity.

E) When receiving positive acknowledgement, receiving side will set SN=VT(S), where VT (A) <= SN< VT(S), and set VT (A) = Smallest SN awaiting acknowledgement.



AM Data transfer: (Receive operations)

Receiving side:

A) The Receiving side should maintain window as well as transmitting:

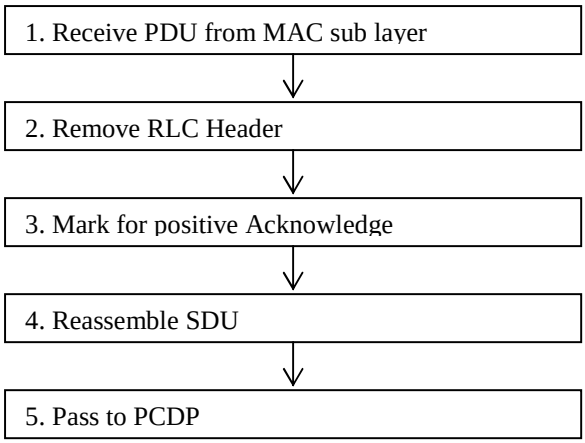
If VR(R) <= SN< VR (MR), SN falls in Receive windows. Otherwise it falls outside.

Receiving side should not send any PDUs with SN that falls outside of this window.

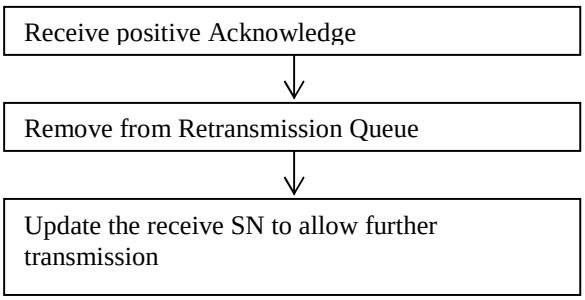
B) After Receiving PDU, receiving side should do the following:

1) Discard received PDU
2) Place PDU in reception buffer
3) Update state variables
4) Reassemble SDU and send it to upper layer
5) Start/stop t-Reordering (in case t-Reordering expires: state variable should be updated and t-Reordering started)

An Overview on Data transmission in Receiving side of AM RLC:

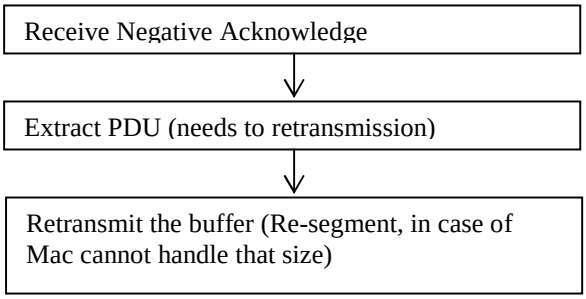


An Overview on receiving positive acknowledgement:



ARQ (Automatic Repeat Request) works in the following way: When Transmitting side of AM RLC receives negative acknowledgement, it demands from transmitting side, having in mind that it saved a copy of all PDUs in re-transmission buffer, to send that PDU again. In case if Mac cannot handle the size, it should make re-segmenting.

An Overview on receiving Negative acknowledgement:



Forming PDU from SDU:

There are two types of PDUs:

1. Data PDU
2. Control PDU (STATUS PDU)

1. Data PDU:

Data PDU consists of 2 parts: pure data which will be fetched from SDU and header for each SDU, which consists of fixed and extended part. Fixed part additionally contains some specifics fields. A set of E and LI comes from each data element which is complete SDU or a portion of it.

AMD PDU Specific Fields:

D/C field: indicates whether the RLC PDU is a RLC data PDU or RLC control PDU (1 bit).

RF field: indicates whether the RLC PDU is an AMD PDU or AMD PDU segment. Re-segmentation Flag (1 bit)

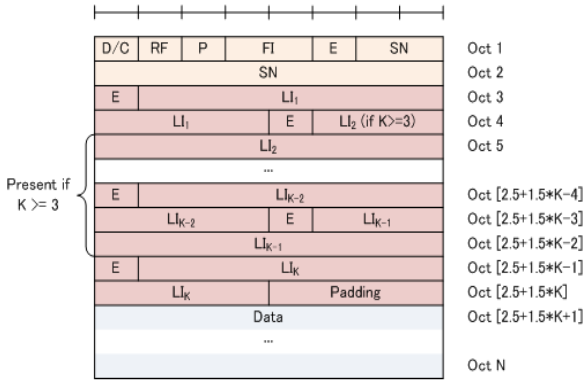
P field: indicates whether the transmitting side of an AM RLC entity **requests a STATUS report from its peer AM RLC entity**.(Polling bit (P) – 1 bit)

SN field represents the sequence number of the corresponding AMD PDU (For an AMD PDU segment, the SN field represents the sequence number of original AMD PDU from which the AMD PDU segment was constructed from).

The sequence number is incremented by one for every AMD PDU.

If we take different data elements from different SDUs and put them in one PDU, one set of E and LI will exist for each data element where E indicates that the following fixed header has one data element with length LI.

For estimating the size of PDU header, the following algorithm has been suggested:



Algorithm for Header size:

X=Number of elements -1

Each AM PDU consists of:

1. Header: which consists of two parts(fixed part and set of Es and LIs) =1.5 X
2. Data

1. fixed ; Size = 16 bits =2 bytes
2. Extension ; Size = 1.5 X
3. padding: ; Size = 0.5 (in case of X is odd)

If (X) is 'odd'
Then **Header size =2.5 + (X) *1.5**
Else **Header size = 2+ (X) *1.5**

2. Control PDU (STATUS PDU)

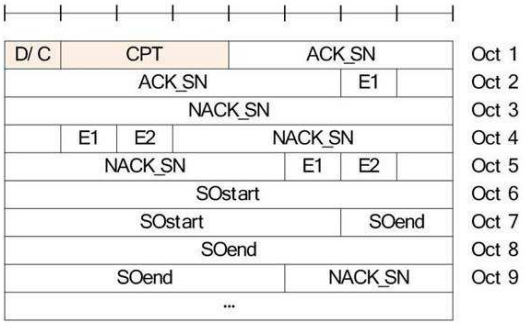
When transmitter sends PDU to its peer via lower layer, receiver should send back Status PDU to indicate whether it received complete data. So, STATUS PDU is used to send acknowledgement for received PDUs. It consists of payload and RLC control PDU header.

The **ACK_SN** field shows SN of the next not received RLC Data PDU, which is not reported lost in the STATUS PDU.

NACK_SN, SOstart, SOend indicate PDU which is lost.

The **CPT** Field represents type of RLC control PDU.

SOstart : The SOstart field represents portion of AMD PDU with SN = NACK_SN that has been detected lost at the receiving side of the AM RLC entity. For estimating the size of STATUS_PDU, the following algorithm has been suggested:



Each STATUS PDU consists of:

1. Header : fixed part including D/C and CPT ; Size = 4 bits
2. Payload: consists of Ack-SN, Nack-SN, E1, E2, possibly SOstart , SOend

1. Fixed part (D/C and CPT); Size = 4 bits
2. Ack _SN + E1 ; Size = 10 + 1

Fixed part ; Size = 15 bits

3. Nack _SN + E1 + E2 +SOstart+ SOend :
; Size: either (10+2) or (10+2+15+15))
.
.
N

Variable part ; Size = N_whole_SN * (10+2) + N_segment * (10+2+15+15))

Status_PDU_size = Round_up to_full_byte(15 + N_whole_SN * (10+2) + N_segment * (10+2+15+15))

Status_PDU_size = RoundUp((15 + N_whole_SN * (10+2) + N_segment * (10+2+15+15)) /8) *8