A Software Architecture for Modular Implementation of Adaptive Protocol Stacks

Marc Schinnenburg, Ralf Pabst, Karsten Klagges, and Bernhard H. Walke
RWTH Aachen University, Faculty 6, Chair of Communication Networks (ComNets), Germany
{msg|pab|kks|walke}@comnets.rwth-aachen.de

Abstract— **This paper presents a modular architecture for the design, implementation and performance evaluation of protocol stacks. The architecture allows a high degree of flexibility through composing complex behavior of a communication system using so-called Functional Units (FUs). The composition of Functional Units (FUs) to form a Functional Unit Network (FUN) is achieved by offering a generic management interface. We tackle the issue of protocol development for communication systems from the perspective of software design and -engineering. The requirements and the resulting interfaces for the Functional Units (FUs) are the main focus of this article. The proposed architecture is presented in the context of a reference model for multi-mode protocol stacks developed in the European Union-funded research project WINNER [1]. An exemplary implementation of the set of Data Link Layer functions as they are currently envisaged in WINNER and an implementation of an IEEE 802.16 (WiMAX) Data Link Layer (DLL) for performance evaluation is presented as a proof of concept.**

Index Terms— **Adaptive Protocol Stack, Flexible Protocol, Multi-Mode, Functional Unit, Functional Unit Network, Modular Protocol**

I. Introduction

NEXT generation wireless communication system requirements target ubiquitous mobile access in a wide variety of application scenarios. These scenarios are differentiated by radio environment, usage, spectrum type or service requirements. In order to meet the full range of requirements such a system requires solutions tailored for the specific scenarios. Even though the individual solutions will be tailored for specific scenarios, a system concept would preferably embrace all these solutions being build on a common technology basis. This implies a flexible, adaptable air interface.

The WINNER project [2] has therefore developed a protocol reference model. The requirements of the above mentioned scenarios are met by different modes in this reference model. The reference model enables the efficient integration of multiple such modes in a complementary way, thereby allowing to take maximum benefit of the commonalities between the modes (see [3]). As a consequence, such a reference model requires protocols that conform to the given structure. In order to exploit commonalities between different modes of operation, the software of these protocols would (i) ideally follow a modular approach to allow a high degree of reusability and (ii) provide suitable structures and interfaces for the flexible composition of the individual modules.

The increasing complexity in today's software systems has led to a number of new methods in software development in the last years. One goal of modern software design is to create systems consisting of preferably small units [4]. Every unit should have one cohesive responsibility, provided through a preferably slim and precise interface. The avoidance of tight coupling and the focus on testability leads to units that are easier to build, to test and to maintain. Thus the development and maintenance costs are reduced. Further, the quality of software is increased. As a second benefit, reusability of units is improved if dependencies between units can be kept to a minimum. The design paradigms from the software design domain can be immediately applied to protocol design for communication systems when a high degree of reusability of the units is targeted at, which is the basis for a flexible protocol architecture. A protocol stack consisting of small and independent units, here called Functional Units (FUs), with cohesive responsibility is therefore one of the key technologies for next generation radio networks.

To show the feasibility and gain a deeper understanding of the presented architecture, the research team of the Chair of Communication Networks (ComNets) has realized an implementation, which is used to model and evaluate new system proposals (e.g. as in WINNER) or evaluate proposals for the extension of existing systems (e.g. as in IST project FIREWORKS, where an Space Division Multiple Access (SDMA) extension has been integrated into the system concept of Worldwide Interoperability for Microwave Access (WiMAX)). Key performance indicators such as throughput per user or per station (e.g. Base Station (BS)) at different layers as well as packet delay at different layers can be evaluated.

The remainder of this paper is organised as follows: Section I-A of the introduction discusses previous and related work. Section II briefly introduces the context of the Modes Convergence Reference Model developed within WINNER. Section III describes the proposed FUs and the different aspects of their interfaces. Section IV outlines possible logical dependencies that may arise between different FUs and how to resolve these. Section V presents a short overview on how different data flows (e.g., in a BS) can be handled in our architecture. Finally, Section VI shows as an example of how the proposed architecture allows to implement the user-plane functions of the DLL

a) as they are currently envisaged by the WINNER project and how this implementation fits into the framework of the WINNER Modes Convergence Reference Model and

[1] (**W**ireless World **In**itiative **Ne**w **R**adio), [1] is part of the 6th Framework Programme for Research

b) of an IEEE 802.16 (WiMAX) system.

The presented implementations serve as protocol stack prototypes for the simulative performance evaluation of the respective systems.

A. Related Work

The protocol architecture proposed in this work is inspired by both i) the work of Siebert and Walke in 2000/2001 (see [5] and [6]) and ii) the work of Berlemann et al. (see [7]). As proposed in these works, we adhere to the approach of complementing generic functionality with system-specific functionality to implement a certain protocol's (i.e., system's) behaviour. The approaches nevertheless differ in some important aspects: [5] and [6] propose rather coarse-grained modules [2] and concentrate on an efficient design of the protocols. Due to the coarse granularity, the solution is relatively monolithic and requires large efforts to enable run-time reconfiguration. [7] and our work focus on the composition of protocol layers out of smaller modules with a single cohesive functionality. The difference between the two is that Berlemann et al. investigate the effects of the composition (i.e., the behaviour of the composed protocols), while our work aims to define a software architecture and generic interfaces to enable the composition of arbitrary modules (FU). Additionally, this architecture offers the desired degree of flexibility to enable the run-time reconfiguration of protocols.

Following up on [5], Sachs in 2003 (see [8]) has taken up the idea of a generic protocol stack in focusing on a generic link layer for the cooperation of different access networks at the level of the DLL. However, not only DLL protocols but also adjacent layers' functions as for instance the control and management of the radio resources as well as mobility have to be considered in a multi-mode capable network. The work of [8] is continued by Koudouridis et al. in 2005 (see [9]) focusing on multi-radio transmission diversity and multi-radio multi-hop networking which can be seen as a use case for the software architecture elaborated in this paper.

Approaches that are related to the one presented in this work also exist in the domain of software engineering for operating systems. Much like [5] and [6], the *x-Kernel* architecture (see [10]) focuses on units that represent entire protocols and defines interfaces to connect them, while this work defines a framework to build up individual protocols from a 'toolbox' of atomic functions.

II. Modes Convergence Reference Model

The Modes Convergence Reference Model as proposed in [3] is designed to facilitate the coexistence and the cooperation of different modes in all logical nodes of the WINNER radio access network[3]. This efficient integration of multiple modes shall be termed 'Modes Convergence'. Modes Convergence-enabled devices will have the ability to dynamically adapt to different modes, giving them maximum flexibility and reducing complexity overhead compared to 'conventional' multi-mode devices (e.g. combined 3G / GSM handsets). Coexistence and Cooperation, i.e., Convergence of modes is desired in the following areas

Convergence in devices: This refers to the case of devices capable of (perhaps simultaneously) operating in different modes at the same time, such as multi-mode UTs, multi-mode BSs or multi-mode RNs. All these devices can benefit from a joint optimization of the functionalities in the different modes. The overall system design also benefits from a convergence of the modes, because the level of commonalities can be kept as high as possible if modes convergence is a design target from the early stages on.

Convergence in spectrum: At the current point in time, the exact spectrum allocation for beyond 3G (B3G) wireless broadband systems such as the WINNER system and the usage of this spectrum by the different envisaged modes or even systems is not known. Thus, the case of devices sharing the same available radio spectrum while operating in different WINNER modes can not be entirely ruled out. These devices could share the available spectrum efficiently if they apply a sharing algorithm that is common to the different modes or maybe common to different systems. Cooperation between modes may increase the overall efficiency of spectrum utilization and may contribute to interference mitigation.

Fig. 1 illustrates the high-level architecture of the multi-mode protocol stack for the flexible WINNER air-interface. The layer-by-layer separation into specific and generic parts (where appropriate) enables a protocol stack for multiple modes in an efficient way: The separation is the result of a design process where the identification and grouping of common (generic) functions is one of the main targets. The generic parts of a layer, marked '-g' in Fig. 1, can be identified on different levels, such as the Physical Layer (PHY), Medium Access Control (MAC) and Logical / Radio Link Control (LLC/RLC) as shown in the figure. The specific parts (marked '-r*n*') are reused in the different modes supported by the protocol stack. The composition of a layer out of generic and specific parts is exemplarily depicted in Fig. 1.

The management and the joint handling of the protocol stack operating in different modes are performed by the stack management, also shown in Fig. 1. When multiple modes are operated, this can be regarded as cross-stack management and it is envisaged to be performed by a stack modes convergence manager (Stack-MCM) which controls the management functionality in the respective protocol layers (N-Layer Modes Convergence Managers, (N)-MCM) in a hierarchical manner. For further reference on the proposed architecture, see [3]. The introduced multi-mode protocol reference model facilitates the structuring of an arbitrary layer into generic and specific parts. In providing guidance for understanding this structuring it marks up optimization potential in questioning the necessity of indicated differences. In this way, an increased protocol convergence is reached enabling an efficient multi-mode capable protocol stack for the WINNER system. This article takes the example of the DLL of the WINNER system to illustrate how such protocols can be composed from a set of mode- independent and mode-specific FUs.

[2]The modules are entire Layer 3 protocol entities, such as Call Control (CC), Mobility Management (MM) etc.

[3]Referred to as logical nodes are (i) the User Terminals (UTs) (ii) the BSs and (iii) the Relay Nodes (RNs)

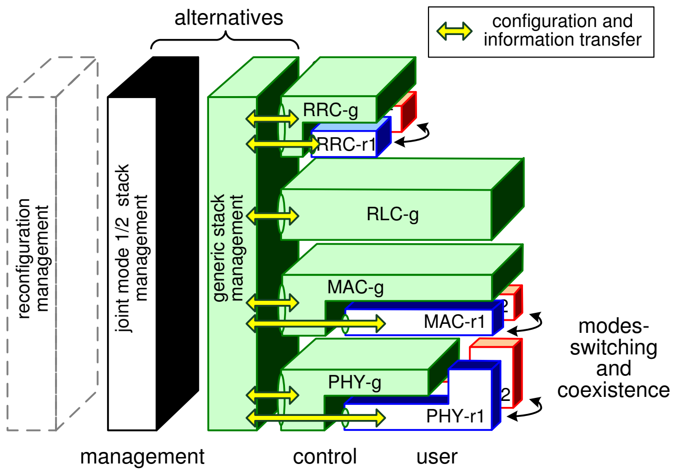

alternatives
configuration and information transfer
reconfiguration management
joint mode 1/2 stack management
generic stack management
RRC-g
RRC-r1
RLC-g
MAC-g
MAC-r1
PHY-g
PHY-r1
2
2
modes-switching and coexistence
management
control
user

Fig. 1. The WINNER multi-mode protocol architecture, facilitating transition (switching) between modes (inter-mode handover) and coexistence of modes (e.g. in relay nodes connecting different modes)

III. Functional Units

As discussed in [3] the DLL of protocol stacks of wireless communication systems in general comprises among others the following set of functions: Automatic Repeat Request (ARQ), Segmentation And Reassembly (SAR), scheduling, multiplexing and buffering. If these units are supposed to be composed and connected in an arbitrary way, the necessity for a generalized interfaces arises. How should FUs be organised to support such a wide range of different tasks? How can these units be connected in a generic way to support the configuration of larger systems based on such units only? To answer these questions, we start analyzing the most fundamental requirements and describe interfaces that allow these requirements to be met. We will describe applications of the defined interfaces and how the FUs can be used to compose complex systems based on these interfaces. The set of functional units together with a description and configuration methodology is intended to form a toolbox for the implementation of protocols.

We have identified four interfaces which are at least necessary to realize an architecture as described in this paper. Namely, these interfaces are:

- Management
- Flow Control
- Data Handling
- Custom

These interfaces of a FU are depicted in Fig. 2. In the following sections these interfaces are described in more depth.

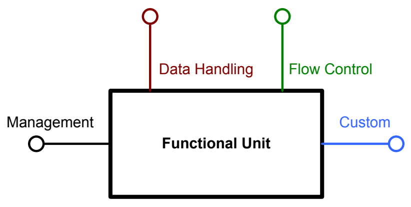


Fig. 2. General requirements of a FU

A. Interfaces

1) Data Handling Interface: The most fundamental requirement for FUs is the ability to handle data. In the following we will denote a basic data unit that is transmitted between FUs a *compound*. For now a compound can be seen as a chunk of data of variable size. FUs as part of a protocol stack may receive compounds for processing before and after such a compound has been transmitted over the air-interface. The first case is called outgoing data flow, while the latter case is referred to as incoming data flow. The interface for handling compounds has to provide services for accepting data in both directions, incoming and outgoing. The interface must further enable the FU to distinguish between compounds of both flows. To support that, it is advisable to choose two different methods: `sendData(Compound)` for compounds in the outgoing flow and `onData(Compound)` for compounds in the incoming flow as depicted in Fig. 3.

2) Flow Control Interface: In practice every FU has only a limited capacity to store compounds and often FUs do not need to store compounds at all to accomplish their task (e.g. forward error correction units). The physical layer on the other hand introduces a bottleneck, limiting the amount of information transmitted and thus the rate at which compounds must be handled. Without any flow control mechanism within a FUN, compounds could leave the FUN with much higher rates than the physical layer could possibly handle. This would result into a dropping of compounds in the physical layer. Buffering between the layers is not an adequate choice either, since the delay between processing the compound in the FUN and data transmission would increase. The increase of delay has several drawbacks.

First, timeout mechanisms would not work as expected. Retransmission timers could lead to retransmission of compounds although the last transmission of these compounds has not even been started. Such compounds would be added to the buffer several times, leading again to increasing delays.

Second, decisions of FUs based on feedback of the physical layer would loose accuracy; and gathered information would be outdated, when the consequences of the decisions would finally manifest.

Thus the need for an intra layer flow control arises. FUs must have the ability to prevent other units from delivering compounds to them, when they decide not to accept additional compounds. There are different reasons for a FU to decide not to accept compounds. All these reasons are direct consequences of the limited resources of the physical layer and thus only apply for *outgoing* flows. Resources in higher layers are usually not a bottleneck for *incoming* flows.

The implementation of this flow control mechanism is realized with two methods

- `isAccepting(Compound)`
- `wakeup()`

Before each `sendData` call the calling FU (whishing to send data) *must* make sure the target FU is accepting via the `isAccepting` method (see Fig. 3). It may also happen that a FU is asked if it is accepting further data and needs in turn to ask the next FU in charge, to return the right result. The `wakeup` of a FU in turn is called by lower FUs to indicate that the lower (calling) FU is accepting data again.

A good example for this is a Stop-and-Wait ARQ protocol: the FU implementing the Stop-And-Wait ARQ is not accepting data if it is waiting for the acknowledgment of a transmission and is accepting data otherwise (assuming there is no buffer built into the ARQ). After the ARQ has received the acknowledgment for the transmission it can change its state from not accepting to accepting again and will in turn call the `wakeup` method of the upper FU to signal its state change. The upper FU can then try to send data to the ARQ again.

Note that flow control is only applied for outgoing flows. Flow control for incoming flows is not necessary. If a FU *A* cannot handle the data in the incoming flow due to limitations of the processing power of the device and if flow control would be available, a FU *B* located below *A* would need to store the data instead. Soon, the buffer of *B* would fill up and the FU below *B* would need to store additional data. Finally, if all FUs are not accepting any data in the incoming flow anymore the physical layer would need to store the incoming data and in the end start dropping incoming packets, too. Therefore, if a device cannot handle the data in the incoming flow, due to whatsoever limitations, respective flow control mechanisms between the sender and receiver need to be established to lower the data rate of the incoming data flow. Additionally, to support variations in data rate of the incoming data flow inherent to the system buffering to some extend can be applied in the FU of concern.

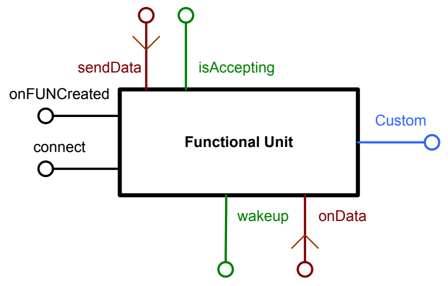


Fig. 3. Detailed interfaces of a FU

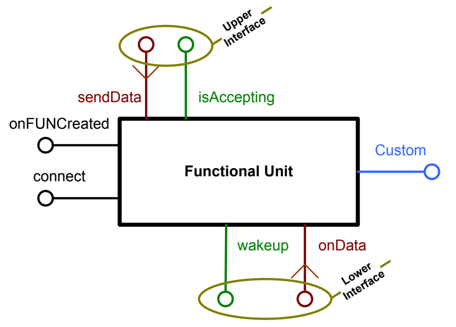
Remark: If no multiplexing and demultiplexing aspects are important, FUs are often drawn in a simplified version where the flow control and data handling interface are replaced by a logical upper and a lower interface (see Fig. 4).

3) Management Interface: The management interface of a FU offers the necessary functionality towards the FU Manager to manage the composition and (re-)configuration of the protocol stack or better a FUN[4]. Since this paper does not focus on the management of FUs, the interface presented in Fig. 3 does not show the complete set of functionality which is currently available. Two methods have been chosen as an example:

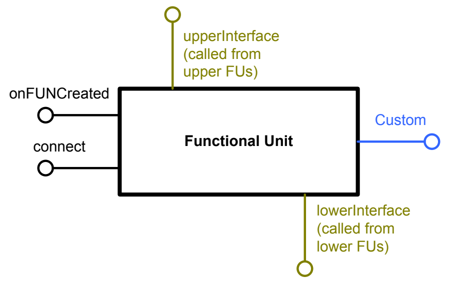
- `connect(FunctionalUnit)`
- `onFUNCreated()`

The `connect` method takes another FU as argument which should be connected to the FU in question. Special containers

[4]see III-B for more details on FUNs. Basically, a FUN is the framework which holds a number of interconnected FUs, forming a protocol with complex behavior



(a) `sendData` and `isAccpeting` can be logically combined into the upper interface of an FU. Accordingly, `onData` and `wakeup` form the lower interface


upperInterface (called from upper FUs)
onFUNCreated
connect
Functional Unit
Custom
lowerInterface (called from lower FUs)

(b) FUs are often drawn only showing upper and lower interface to leave out unnecessary details

Fig. 4. FU with simplified (logical) interfaces

to support multiplexing and demultiplexing of data when the protocol stack is in full operation are maintained by this method.

`onFUNCreated` is a hook which is called by the surrounding framework of a FU to signal the successful creation of a FUN to the FU. Any special tasks the FU may need to undertake to get into a proper state for operation can be handled here. E.g. the FU can access other FUs of the FUN to resolve any dependencies it may have.

4) Custom Interface: All aforementioned interfaces are generic interfaces of a FU which need to be supported by each FU in order to ensure proper working as part of the described framework. However, it can be beneficial for FUs in terms of system performance (optimization aspects) to offer additional interfaces. These interfaces are summarized under the Custom interface.

The generic interfaces (described so far) don't export much information about the internal state of a FU (except whether a FU is accepting data or not through the `isAccepting` method). This contributes to one of the design targets formulated at the beginning: reduced coupling between FUs to increase reusability. For a further discussion on reusability and dependencies see section IV.

Fig. 5 and Fig. 6 show two examples for FUs that can have additional (custom) interfaces.

The method `retransmissionsPending` of the ARQ or `getLength` of the buffer can be used by other FUs of the FUN to optimize their operation. E.g. a FU can prefer a certain buffer over others if its length [5] exceeds a certain threshold.

[5]length is used synonym to size or fill level here

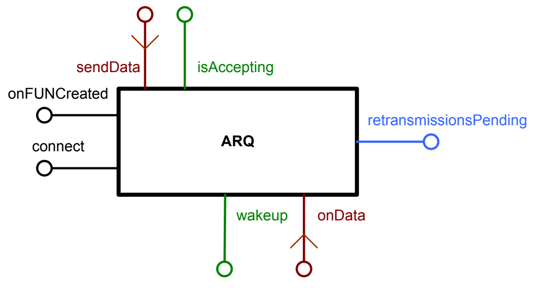


Fig. 5. FU with additional custom interface: ARQ

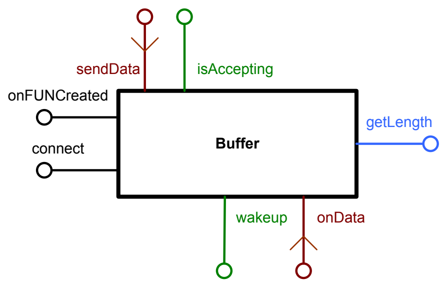


Fig. 6. FU with additional custom interface: Buffer

B. *Functional Unit Networks*

The methods `sendData` and `onData` are called by other FUs to propagate compounds through a Functional Unit Network (FUN). A simple setup of a FUN including the FU Manager is shown in Fig. 7. Three FUs are connected by the FU Manager to form a FUN. The outgoing data flow is from top to bottom (thus higher layers to lower layers). Every FU would forward the received data to its next lower FU without any choice. No (de)multiplexing aspects are present in this FUN. This is however only rarely the case within a protocol stack. Thus the frameworks needs means to handle (de)multiplexing.

Fig. 8 shows a part of a FUN with five FUs: two at the top, two at the bottom and one in the middle at a higher level of detail). The FU in the middle now needs to able to

a) multiplex outgoing data from upper FUs
b) demultiplex incoming data to upper FUs
c) demultiplex outgoing data to lower FUs.

Therefor, every FU contains three strategies for (de)multiplexing and according sets of references to other FUs: The Connector, the Deliverer and the Receptor (see Fig. 8). FUs call the `sendData` method of other FUs in their connector set to pass on compounds in the outgoing flow and call `onData` of FUs in their deliverer set to pass on compounds in the incoming flow. The strategy of Deliverer and Connector determine which FU will actually receive the compound. To wakeup upper FUs (see section III-A.2) the calls the Receptor which again forwards the call according to its strategy.

The FUs can be connected to multiple units in both directions to support multiplexing and scheduling, realized by choosing different strategies to select a unit for compound delivery. A FUN can now be constructed by choosing FUs from a toolbox of FUs and connecting them, defining their

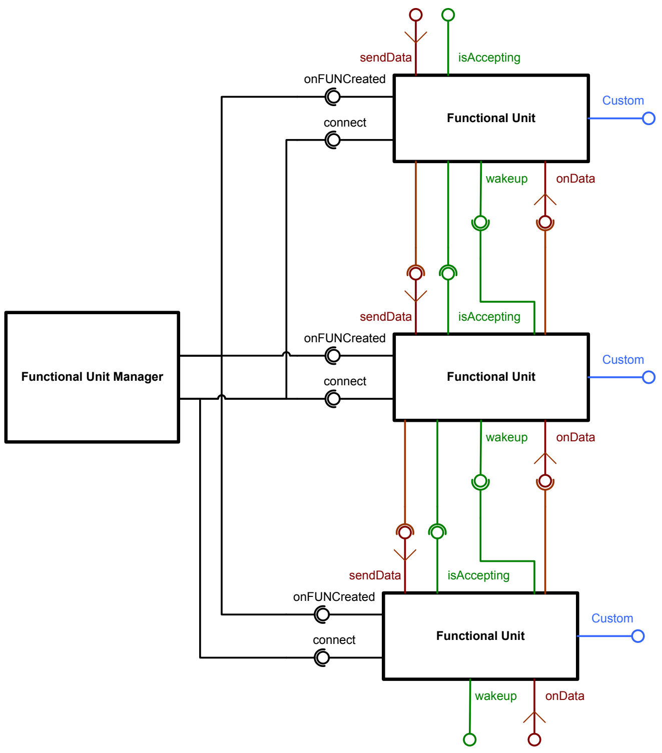


Fig. 7. Composition of FUs to form a simple FUN

connector and deliverer sets. It is possible to further identify a set of units as sink for outgoing flows: Compounds delivered to these units are leaving the FUN for delivery to lower layers. Another set of units may be identified as sink for incoming flows: Compounds delivered to these units are leaving the FUN for delivery to higher layers. Consequently, a FUN can be seen as a bi-directional data processing network. Input to the network is injected using either the `onData` or `sendData` method of any of the FUs. The output of the network is measurable at the sink units. The communication between peer entities of a protocol can be understood as data exchange between two FUNs.

C. *Commands*

Whenever a compound arrives in a FU, the FU gains control over the compound and can realize different behaviors by handling the compound accordingly. It may choose to mutate or drop the data unit, buffer it, forward it to other FUs or inject new data units into the FUN. A large class of FUs is characterized by enriching the compound, adding control information on outgoing compounds and reinterpreting the added information on incoming compounds. Usually these FUs provide a transparent connection to other FUs above. An ARQ protocol for example adds sequence numbers as control information to the compounds of the outgoing flow. It creates and injects compounds as acknowledgments in order to reply to compounds of the incoming flow. The ARQ instance in the peer FUN reinterprets the added control information, delivers valid information frames to some FU in the deliverer set and consumes dedicated compounds containing acknowledgments.

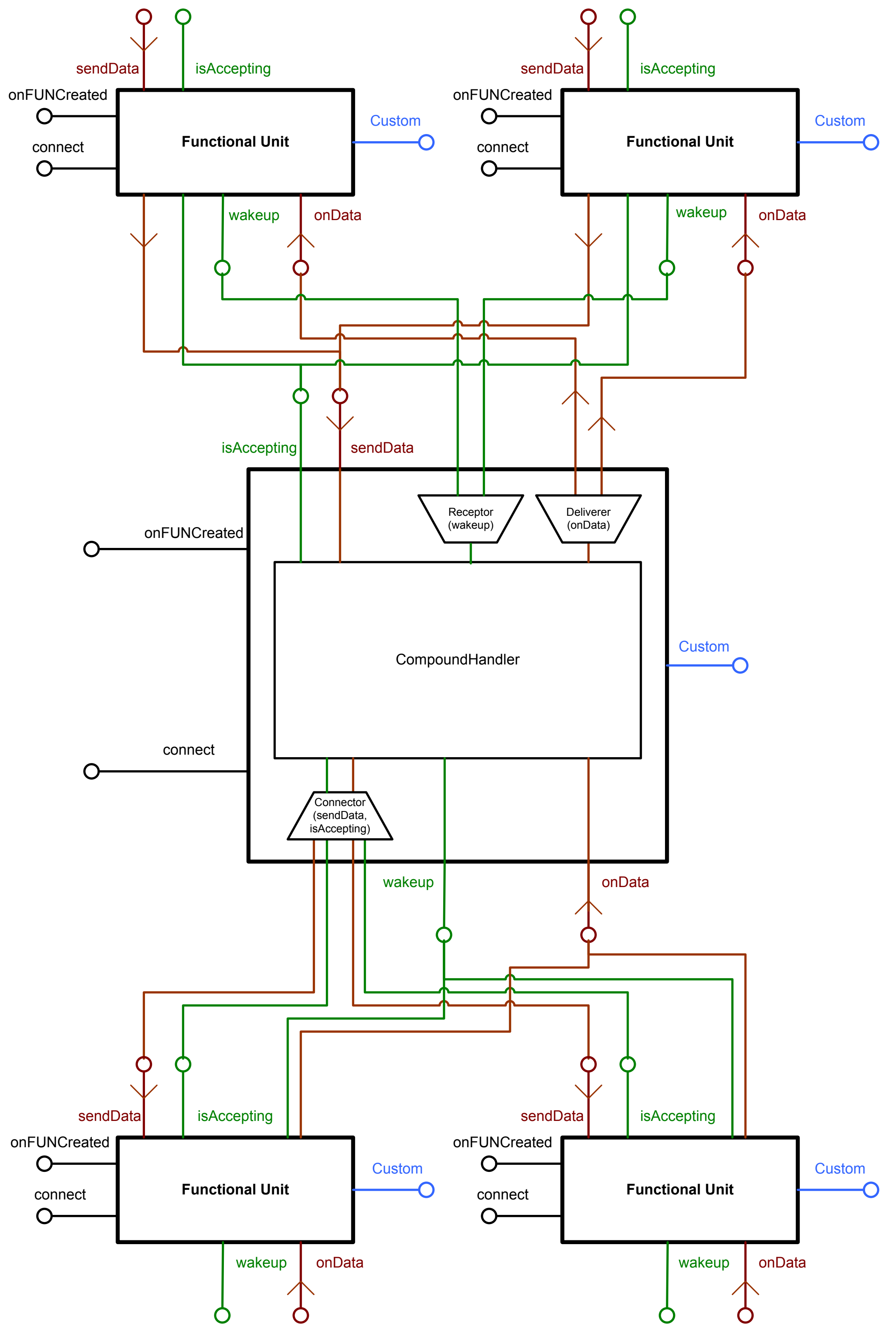


Fig. 8. FUN with FU at full level of detail

The control information added by FUs is called *command*. The command can have different characteristics for different purposes, like an information command or an acknowledgment command for the ARQ. The ARQ in our example is completely invisible to the FUs above. Even underlying FUs do not need to have knowledge about the control information added by ARQ implementations. The only FU that is required to be able to handle the ARQ command is the peer unit of the ARQ. Sometimes, however, FUs add commands that are important to other FUs either in the peer FUN or within the same FUN. Connection identifiers may serve as an example for such information. FUs may require being able to retrieve the destination address of a compound which is part of a higher level (another FUs) routing command.

This leads to the requirement of having a possibility to access commands added by other FUs. Note that FUs cannot simply reinterpret control information added by other units to the compounds' data. FUs have no information about the layout of the FUN and therefore also have no information about the layout of the combined control information within the compound. There might be an arbitrary number of FUs in between the unit that added the control information and the unit that intends to access it. Additionally, the data might have been heavily modified by other FUs in between. The solution is to attach a set of commands to each compound. Since a FUN has a known number of connected FUs, there is a known set of potential commands.

The set containing all the commands of every FU within a FUN is called *command pool*. We can now specify more precisely that the *compound* as defined in Section III-A.1 is the union of a data unit and a command pool. Initially all commands within the command pool of a compound are inactive. The data attached to a compound is set to the data unit delivered by higher layers for transmission. A data unit is initially empty for compounds being created/injected in the FUN (e.g., ARQ acknowledgements). Parts of the command pool get activated during the propagation of a compound through the FUN, where every FU activates its command when in control. At the same time FUs can mutate the data.

A set of activated commands ordered by their time of activation is named a *command sequence*. A FUN is required to be free of cycles to assure that commands do not get activated more than once. Hence, an unambiguous command sequence must exist in which single commands may be retrieved.

1) Relaying of Compounds: The activation of commands from a non-extendable command pool introduces the problem of implementing relaying FUs [11]. Having a single point of activation implies that compounds may not cross the borders between the two FUNs from incoming to outgoing data flows. No FU may forward a compound using `sendData`, when it received the same compound via `onData`. This is a direct consequence from the activation of commands. Otherwise, it would be possible for the compound to be delivered to a FU that already activated its command. To implement relaying, the relaying FU has to inject a copy of the received compound, which has only those commands activated and copied that are in the command sequence before the relaying unit. The rest of the commands will remain inactivated. Fig. 9 gives an overview about the activation status of commands within a compound as it is passed from the originating FUN (left) via a FUN that has relaying capabilities (middle) to the final destination FUN (right). The relaying FU injects a copy of the incoming compound which holds a copy of the activated command of FU 'A' (and activates its own command 'B'), because the compound is not going to pass through 'A' in the outgouing path of the relaying FUN. Note that the relaying FU 'B' in Fig. 9 is transparent in the case of outgoing flows (left), for incoming compounds it can either perform the relaying as explained above (center) or decide not to further relay the compound because it has reached its final destination (right).

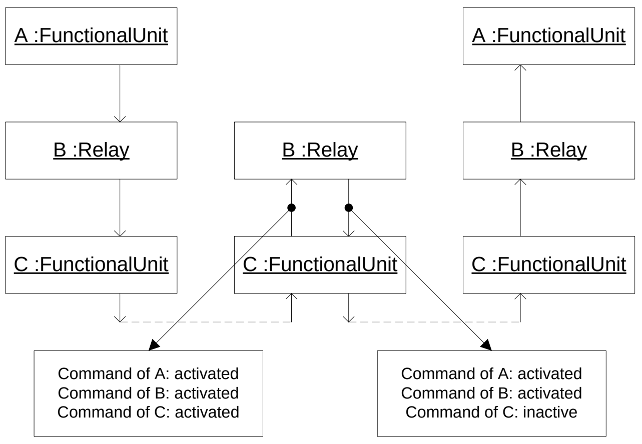


Fig. 9. Partial copy of a command pool for relaying

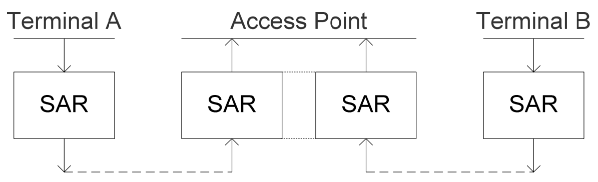
2) Coding of Commands: Besides commands being accessible by other FUs, delaying the coding of commands as part of the data has another advantage. Often information in communication protocols is not transmitted explicitly as a stream of bits, but implicitly through the choice of radio resources element like time, frequency, space or code. In a Time Division Multiple Access (TDMA) system for example with fixed slot reservations for connections, it would be useless to explicitly transmit connection identifiers. Nevertheless the information is indirectly transmitted through the choice of a specific slot. Such a slot must be chosen at some point of time based on the connection identity. A command provided by a connection aware FU may contain the connection identifier. But the choice how to transmit the connection identity is delayed, and the outcome may be different depending on the system. As we have shown, attributes of commands serve different purposes. Some are meant to be transmitted, while other attributes are only meaningful within a FUN and are not meant to be sent to the peer FUN. In order to be explicit about the purpose of a command attribute, we divide the attributes of commands in two distinct sets: The local on the one hand and the peer set on the other hand.

D. Five Aspects of a Functional Unit

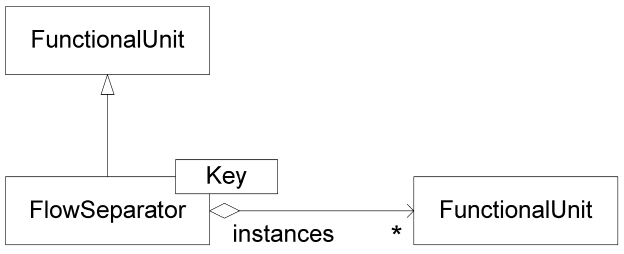
To summarize the discussion above, we distinguish five aspects of a FU:

1) **Compound Handler:** implements the handling of compounds (Data Handling Interface) of a FU including intra FUN flow control (Flow Control Interface). The methods provided are
   - `onData(Compound)`,
   - `sendData(Compound)`,
   - `wakeup()` and
   - `isAccepting(Compound)` $\longrightarrow$ `Boolean`.

   Handling of compounds includes mutation, dropping, injection and forwarding. Activation and initialization of commands is considered as mutation.



(a) FUs of different flows



(b) UML Diagram for FlowSeparator in a FUN

Fig. 10. Flow Separation and dynamic instantiation of Functional Units

2) **Command Type Specifier:** defines the type of command provided by the FU. This type will be used to create an initial command pool and to verify unit dependencies as will be discussed in Section IV
3) **Connector:** holds the set of FUs that compounds will be delivered to in the outgoing direction; defines a strategy to select the appropriate FU for a given compound.
4) **Receptor:** holds the set of FUs in which the FU itself is in the connector set; defines a strategy to trigger outgoing data from the set of previous FUs (wakeup).
5) **Deliverer:** holds the set of FUs that compounds will be delivered to in the incoming direction; defines a strategy to select the appropriate FU to deliver a given compound.

IV. Unit Dependencies

Ideally a FUN would consist of FUs without any inter-unit dependencies. But that is not an option for building real world protocol stacks. Knowing what kinds of unit dependencies exist, what they imply and when to accept them is essential for the design of FUs and FUNs. We distinguish between two different kinds of unit dependencies: Direct and deferred coupling. **Direct coupling** is a dependency on the Custom *interface* of another FU (see section III-A.4); **deferred coupling** is the dependency on the *command* of another FU. When FU 'A' depends on the interface or the command of FU 'B' we say that 'B' is a friend of 'A'. Direct dependencies arise for example for

- multiplexing FUs that need assistance of their friends below them to decide where to deliver compounds.
- horizontal collaboration; FUs responsible for realising control plane functionality, receiving supervisory frames from a peer node and configuring their friends to modify the user data plane accordingly.
- vertical collaboration; layered protocol functions that must work close together but change behaviour in different places of a protocol stack.

To avoid tight coupling, those dependencies should rely on the most general interface possible [4]. The goal should be to make FUs depend on families of units sharing a common interface, and not to depend on a single type of FU. This allows friends to be exchanged without modifying the dependent FU. Since the exact layout of an FUN is unknown to the FUs, the FUN provides services for the FUs to find their friends by name and desired interface. Making the names of the friends a configuration option to dependent FUs results in a high degree of flexibility. Friends can be retrieved once after reconfiguration of the FUN. To retrieve a command from a command pool, the retrieving FU does not need to rely directly on an interface of the command's provider. It relies on the command's provider to be present in the FUN and on the type of command the provider specified.

There is another subtle difference between the direct coupling using the custom interface of a FU and the deferred coupling using the command attached to a compound. The interface, if used for information retrieval, yields a result which reflects the current state of the FU. The deferred coupling, using a command attached to the compound for, can only offer information reflecting the state of a FU at the time the compound passed through it.

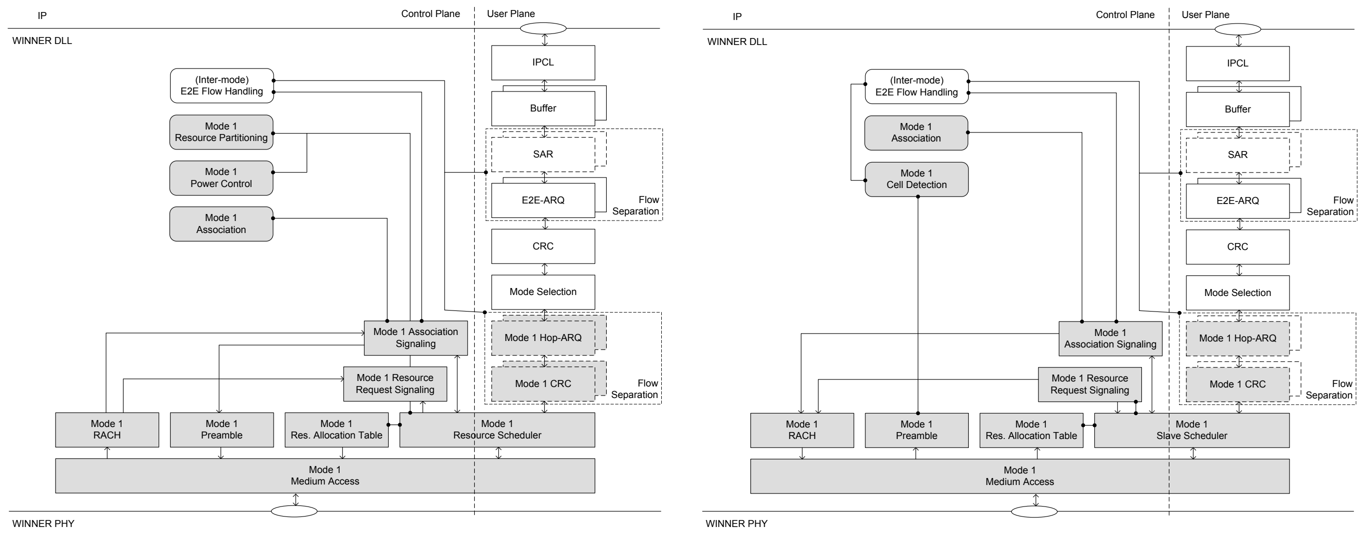
V. Flow Separation

FUs as described can have one or more internal states and exhibit a related behavior. A SAR unit for example needs to store segments of compounds to be able to apply segmentation and reassembly. A FUN therefore obviously requires different instances of a SAR unit for different peer FUNs as depicted in Fig. 10(a). To dynamically create FUs within a FUN depending on their data flow a so-called *flow separator* is proposed (see Fig. 10(b)). The flow separator itself is a FU, configured by a key to distinguish flows and a strategy to create FU instances. It is based on the *Instantiator* design pattern proposed by Gamma et al. in [12]. The compound handling including flow control is delegated to the according instance by the flow separator. This way, not each FU is burdened with task to be able to handle multiple flows, but a specialist was created that can be reused.

VI. Proof of Concept

A. Winner Protocol Stack

Fig. 11 exemplarily shows how the intended functionality of the DLL as it is currently discussed by the WINNER project can be composed out of a set of mode-independent FUs (white) and a number of mode-specific FUs (grey). An implementation of the DLL based on this concept is available as part of Wireless Network Simulator (WNS) for performance evaluation of the WINNER system. The DLL is depicted here only for one mode (Mode 1). However, in the WINNER project at least 2 modes are currently envisaged. It should



(a) in the Base Station (b) in the User Terminal

Fig. 11. WINNER DLL as a Functional Unit Network

be noted that the mode-specific behaviour of the FUs shown in gray is achieved by pure parameterization, while identical implementations are being used when Mode 2 is used instead of Mode 1.

The used FUs can be further subdivided into three different classes:

1) The common, system-independent functions, these can be taken from a toolbox of generic protocol functions that can also be used to implement protocols for other, non-WINNER radio systems, examples are
   - ARQ units. The figure shows an E2E-ARQ for securing packets end-to-end over multiple radio hops and a Hop-ARQ that operates on a per-hop basis.
   - Buffers
   - Segmentation and Reassembly Units
2) The mode-independent, but WINNER-specific functions
   - IP Convergence Layer (IPCL)
   - Mode Selection
3) The WINNER-mode-specific functional units, which are shown as grey boxes. Example for such functions are
   - Resource Scheduler, which needs have detailed knowledge of the physical layer mode being used.
   - Hop-ARQ, which is likely to be a Hybrid ARQ and thus also closely linked to the physical layer mode being used.

The change of the parameterization of the mode-specific FUs to configure a new mode would be the responsibility of the entity that manages the respective protocol layer (see Fig. 1).

In addition to the discussed FUs Fig. 11 shows so-called Layer Services (drawn as boxes with round corners). These services, in contrast to the FUs, don't directly exchange information with some peer service in another protocol stack. They control, use and observe FUs within the protocol stack to fulfill a certain task. An example is the "Mode 1 Cell Detection" which simply observes the reception of Mode 1 Preambles by the respective FU. In a publish-subscribe manner the "Mode 1 Cell Detection" is subscribed to the "Mode 1 Preamble" reception. Again, this promotes minimizing dependencies between the modules of a layer to enhance reusablity (any other additional service or FU can subscribe to the "Mode 1 Preamble" reception) and flexibility (the strategy for the cell detection can easily be exchanged).

B. IEEE 802.16 Protocol Stack

The presented FU concept has also been used for an implementation of the IEEE 802.16 protocol stack. Fig. 12 shows the Functional Unit Network of an 802.16 protocol stack which realizes the BS and the Subscriber Station (SS) data plane and the OFDM PHY layer. As in the WINNER protocol stack, the 802.16 implementation uses several common FUs like the ARQ or the SAR. These generic FUs are drawn with white boxes. FUs that are modified or created for the FUN are drawn in grey boxes.

The function of the FUN can be divided into two classes, namely the Connection Control and the Radio Resource Control.

1) *Connection Control*
   The Classifier classifies compounds realizing flow separation for buffers, ARQ and SAR entities per CID. The Relay Mapper performs a rewrite of CIDs in the relay station for pairs of connections and a partial copy of the compound as shown in Fig. 9. As a result the relay implementation is not equipped with the FUs above the Relay Mapper except the buffers.
2) *Radio Resource Control*
   Radio Resource Management is mainly controlled by the Frame Builder which provides a common framework for frame based protocols. The frame is indirectly specified through the FUs that are directly attached to the upper side of the Frame Builder and through the

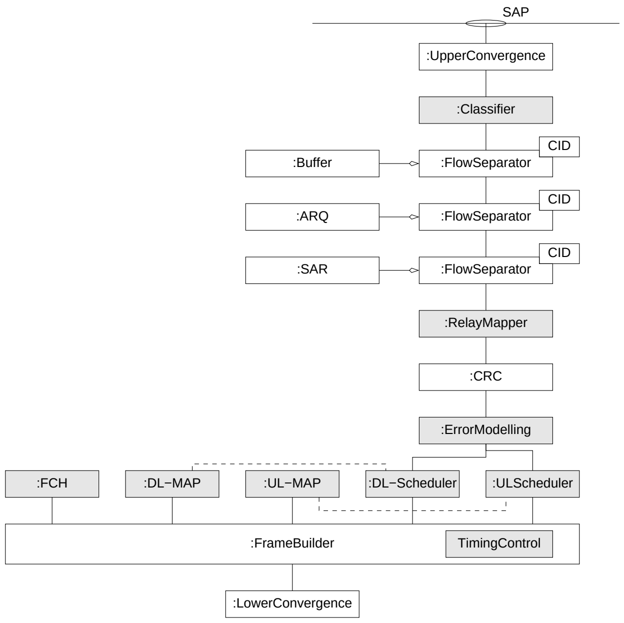


Fig. 12. Functional Unit Network of the IEEE 802.16 BS and SS

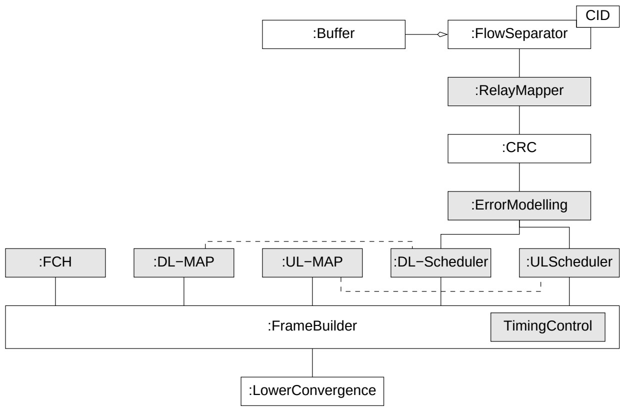


Fig. 13. Functional Unit Network of the IEEE 802.16 RS

Timing Control which is part of the Frame Builder. The Timing Control activates the attached FUs in a previously specified way. While the Frame Builder is a generic component, the Timing Control is system specific.

VII. Conclusion

We presented a software framework for flexibly building protocol stacks out of Functional Units (FUs). An implementation of this framework is used to evaluate the performance of wireless communication systems by means simulative performance evaluation. Simple, cohesive FUs are connected using a uniform interface to form arbitrary FUNs in order to build complex protocols. An interface, divided into the following five aspects, has been defined: Compound Handler, Command Type Specifier, Connector, Receptor and Deliverer. A solution for keeping dependencies between FUs at a minimum level is presented. The problem of flow separation is briefly discussed and a solution is sketched.

The presented framework opens up potential to

- Exploit similarities between the protocols of different wireless systems and hence facilitate efficient implementation of reconfigurable multi-radio devices
- Build systems based on multiple modes, each optimized to suit a specific environment, without the need to provide complete protocol software for each mode in the terminals and base stations.
- Implement software-defined radios where not only the physical layer but also higher layer protocols can be adapted through parameterization and configuration.
- Accelerate protocol stack development and performance evaluation.

Future work includes further investigations on the identification and implementation of reusable FUs conforming to the presented interface and on the management, parameterization and seamless reconfiguration of protocols represented through FUNs. Additionally, it is planned to release an implementation of the presented software architecture within the European research project *openWNS*, which currently applies for funding as part of the FP7. The *openWNS* project will release an open source system level simulator for the evaluation of wireless communication systems.

References

[1] http://www.ist-winner.org/.
[2] W. Mohr, “The winner (wireless world initiative new radio) project - development of a radio interface for systems beyond 3g.” in *Proc. of IEEE Personal Indoor and Mobile Radio Conference 2005 (PIMRC05)*, Berlin, Germany, Sep 2005.
[3] L. Berlemann, R. Pabst, M. Schinnenburg, and B. Walke, “A flexible protocol stack for multi-mode convergence in a relay-based radio network architecture,” in *Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005. IEEE 16th International Symposium on*, vol. 2, 11-14 Sept. 2005, pp. 769–774Vol.2.
[4] H. Sutter and A. Alexandrescu, *C++ Coding Standards*. Addison-Wesley, 2005.
[5] M. Siebert, “Design of a generic protocol stack for an adaptive terminal,” in *Proc. of 1st Karlsruhe Workshop on Software Radios*, Karlsruhe, Germany, Mar 2000, pp. 31–34.
[6] M. Siebert and B. Walke, “Design of generic and adaptive protocol software (dgaps),” in *Third Generation Wireless and Beyond (3Gwireless ’01)*, vol. 0, San Francisco, US, Jun 2001. [Online]. Available: http://www.comnets.rwth-aachen.de
[7] L. Berlemann, R. Pabst, and B. Walke, “Multimode communication protocols enabling reconfigurable radios,” *EURASIP Journal in Wireless Communications and Networking, Special Issue: Reconfigurable Radio for Future Generation Wireless Systems*, vol. 2005, no. 3, pp. 390–400, Aug 2005.
[8] J. Sachs, “A generic link layer for future generation wireless networking,” in *Communications, 2003. ICC ’03. IEEE International Conference on*, vol. 2, 11-15 May 2003, pp. 834–838vol.2.
[9] G. P. Koudouridis, R. Agüero, E. Alexandri, and et al., “Generic link layer functionality for multi-radio access networks,” in *Proc. of 14th IST Mobile & Wireless Communications Summit*, Dresden, Germany, Jun 2005.
[10] N. Hutchinson and L. Peterson, “The x-kernel: An architecture for implementing network protocols,” *Software Engineering, IEEE Transactions on*, vol. 17, no. 1, pp. 64–76, Jan. 1991.
[11] R. Pabst, B. Walke, D. C. Schultz, and et al., “Relay-based deployment concepts for wireless and mobile broadband radio,” *IEEE Communications Magazine*, pp. 80–89, Sep 2004.
[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, professional ed. Boston: Addison-Wesley, 1994.