Implementing the S1 Application Protocol in a Cloud Radio Access Network Environment

Keijo Lehtinen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 3.8.2016

Thesis supervisor:

Prof. Riku Jäntti

Thesis advisor:

D.Sc. (Tech.) Kalle Ruttik

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Keijo Lehtinen

Title: Implementing the S1 Application Protocol in a Cloud Radio Access Network Environment

Date: 3.8.2016 Language: English Number of pages: 7+58

Department of Communications and Networking

Professorship: Networking Technology

Supervisor: Prof. Riku Jäntti

Advisor: D.Sc. (Tech.) Kalle Ruttik

Cloud Radio Access Network (C-RAN) development is one branch of fifth generation (5G) mobile network research. In C-RAN, most processing related to network operation is moved from the base stations to data center servers, thus providing flexibility and cost savings.

The Aalto Radio Framework (ARF) software implements part of the Long Term Evolution (LTE) specification. It includes a partial implementation of a base station that implements the air interface. In LTE, a base station uses the S1 interface to communicate with the network core. The control plane of this S1 interface uses the S1 Application Protocol (S1AP). In this work, the S1AP is fully implemented as an independent module, which is then integrated into the ARF software.

The S1AP is specified using Abstract Syntax Notation One (ASN.1). To implement a protocol specified with ASN.1, an ASN.1 compiler is used to generate encoder and decoder code from the abstract syntax. In this work, the open source asn1c compiler was used. However, the asn1c compiler does not support ASN.1 information object classes. This causes an incompatibility between asn1c and the S1AP abstract syntax, and workarounds had to be developed. This involved modifying part of the S1AP abstract syntax without altering the original meaning, and writing scripts to generate code from asn1c incompatible parts of the resulting abstract syntax.

An (API) was also developed to provide a much easier to use interface to use S1AP without requiring knowledge of ASN.1. Use of this API was integrated into ARF to enable and facilitate further development into the S1 interface logic.

Keywords: S1, S1AP, C-RAN, ASN.1, 4G, 5G, asn1c

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Keijo Lehtinen

Työn nimi: S1 applikaatio protokollan implementointi radioverkon pilvi toteutuksessa

Päivämäärä: 3.8.2016 Kieli: Englanti Sivumäärä: 7+58

Tietoliikenne- ja tietoverkkotekniikan laitos

Professuuri: Tietoverkkotekniikka

Työn valvoja: Prof. Riku Jäntti

Työn ohjaaja: TkT Kalle Ruttik

Radioverkon pilvitoteutuksen tutkimus on osa viidennen sukupolven mobiiliverkkojen tutkimusta. Radioverkon pilvitoteutuksessa suurin osa tukiasemien verkon toimintaan liittyvästä prosessoinnista pyritään siirtämään datakeskuksiin. Näin saavutetaan joustavuutta verkon toteutukseen ja kustannukset laskevat.
Aalto Radio Framework (ARF) on ohjelmisto, joka toteuttaa osan LTE verkon toiminnasta. Se sisältää osittaisen toteutuksen tukiasemasta joka toteuttaa radio rajapinnan. LTE verkossa tukiasema keskustelee verkon ytimen kanssa S1 rajapinnan avulla. Tämän rajapinnan kontrollitaso käyttää S1AP protokollaa. Tässä diplomityössä toteutetaan kyseinen protokolla kokonaisuudessaan ja integroidaan se osaksi ARF ohjelmistoa.
S1AP protokolla on määritelty Abstract Syntax Notation One (ASN.1) notaation avulla. ASN.1:n avulla määritelty protokolla voidaan toteuttaa ASN.1 kääntäjän avustuksella. ASN.1 kääntäjä muuntaa abstraktin notaation ohjelmakoodiksi, joka koodaa ja dekoodaa protokollan viestejä. Tässä diplomityössä on käytetty avoimen lähdekoodin ASN.1 kääntäjää nimeltä asn1c. Asn1c ei kuitenkaan täysin tue ASN.1:n informaatio luokka konseptia. Tämä aiheuttaa yhteensopimattomuuden S1AP:n abstraktin syntaksin ja asn1c kääntäjän välille. Tämän ongelman ratkaisemiseksi S1AP:n abstraktia syntaksia on muokattu asn1c yhteensopivampaan muotoon siten, ettei syntaksin alkuperäinen merkitys kuitenkaan muutu. Lisäksi on toteutettu skriptejä jotka jäsentävät osan tästä abstraktista syntaksista ja tuottavat ohjelmakoodia sen perusteella.
Lisäksi tässä työssä toteutettiin uusi ohjelmisto rajapinta, joka tarjoaa helppokäyttöisen käyttöliittymän S1AP:n käyttämiseen. Tämä rajapinta ei myöskään vaadi ASN.1:n tuntemusta. Lopuksi S1AP toteutus integroitiin ARF ohjelmistoon, joka mahdollistaa ja helpottaa S1 rajapinnan toiminnallisen logiikan jatkokehitystä.

Avainsanat: S1, S1AP, C-RAN, ASN.1, 4G, 5G, asn1c

Preface

I would like to thank both my thesis supervisor Prof. Riku Jäntti and advisor D.Sc. (Tech.) Kalle Ruttik for giving me the opportunity to work on this interesting project. Their guidance and support was a great help and kept me motivated even during the writing process. I would also like to thank my family for their patience and always giving me the encouragement I needed.

Otaniemi, 3.8.2016

Keijo T. Lehtinen

Contents

Abbreviations

3GPP	3rd Generation Partnership Project
4G	Fourth generation
5G	Fifth generation
AP	Application Protocol
AP ID	Application Protocol Identity
API	Application Programming Interface
ARF	Aalto Radio Framework
ASN.1	Abstract Syntax Notation One
AuC	Authentication Center
BER	Basic Encoding Rules
C-RAN	Cloud Radio Access Network
CCITT	Consultative Committee for International Telephony and Telegraphy
CER	Canonical Encoding Rules
DER	Distinguished Encoding Rules
E-UTRAN	Evolved Universal Terrestrial Radio Access Network
eNodeB	E-UTRAN Node B
EPC	Evolved Packet Core
GPRS	General Packet Radio Service
GTP	GPRS Tunnelling Protocol
GTP-U	GPRS Tunnelling Protocol User Plane
HLR	Home Location Register
HSS	Home Subscriber Server
IANA	Internet Assigned Numbers Authority
IE	Information Element
IETF	Internet Engineering Task Force
IP	Internet Protocol
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
LTE	Long-Term Evolution
MME	Mobility Management Entity
NAS	Non Access Stratum
P-GW	PDN Gateway
PDN	Public Data Network
PDU	Protocol Data Unit
PER	Packed Encoding Rules
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RFC	Request for Comments
RRU	Remote Radio Unit
S-GW	Serving Gateway
S1-MME	S1 Control Plane Interface
S1-U	S1 User Plane Interface
S1AP	S1 Application Protocol

SAE	System Architecture Evolution
SCTP	Stream Control Transmission Protocol
SDR	Software Defined Radio
TA	Tracking Area
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
UPE	User Plane Entity
X2AP	X2 Application Protocol
XER	XML Encoding Rules
XML	Extensible Markup Language

1 Introduction

Mobile networks have become an integral part of our everyday lives during the last few decades and their influence is only growing as technology keeps improving. We are currently using fourth generation (4G) mobile networks and rapidly developing new and improved techniques to become the fifth generation (5G) mobile network. The main goals for 5G mobile networks are large improvements in data rates, significantly reduced latency, improved coverage and enhanced spectral and signalling efficiency. In addition, a massive amount of simultaneous connections will be supported. This will be required in particular to support the Internet of Things (IoT) use case where physical objects are equipped with network connectivity.

A big part of 5G research is done by developing improvements on top of 4G Long-Term Evolution (LTE) networks. One such improvement is the Cloud Radio Access Network (C-RAN) concept. Traditional LTE network architecture is formed by base stations, which connect to the Evolved Packet Core (EPC). Connectivity to external packet data networks is provided through the EPC. In C-RAN, much of the functionality of base stations is moved to cloud computing infrastructure. The base station is left with a remote radio unit (RRU), which is connected to the the cloud with a high speed data link. Moving the baseband processing from the base station to the cloud, where it can be done centrally, brings many benefits. The most important is the simplification of the base station. They become cheaper and require much less energy to operate. This in turn enables the large scale deployment of small cells to improve network capacity. Centralized processing also brings flexibility in coordination among cells enabling efficient use of resources. The network topology can be reconfigured on the fly to dynamically adapt to changing circumstances or equipment failures.

Introducing software defined radio (SDR) on the cloud computing side can bring additional benefits. Baseband processing is a timing sensitive real-time function. By using a SDR implementation that can run on general purpose processors, it becomes possible to avoid the special purpose hardware traditionally used in baseband processing. This results in significant reduction in hardware costs, but increased difficulty to fulfil timing related requirements.

The Aalto Radio Framework (ARF) platform is a network testbed being developed by the Aalto C-RAN research group. It brings together SDR and cloud processing. It can operate over the air as a radio network or as a simulation platform. The testbed is designed for research of radio resource management in mobile networks. To enable this, the testbed is implemented in a modular architecture that separates radio interface management from process scheduling and management.

Currently, the ARF platform contains mostly functionality relating to the radio communication between base station and user equipment (UE). Functionality where the base station implementation communicates with the EPC is missing. This functionality needs to be added to enable further development of the ARF software towards a more complete base station implementation. In LTE network architecture, a base station communicates with the EPC through the S1 interface, which separates control plane and user plane traffic. The control plane traffic is sent to a node called

Mobility Management Entity (MME) in the EPC. This control plane traffic uses the S1 Application Protocol (S1AP). Implementing this control plane communication capability would be a logical first step in the further development of the ARF platform.

The goal of this work is to implement S1AP into the ARF software. This includes the ability to generate, encode, decode, send and receive S1AP messages. The S1AP implementation should provide an easy to use communication channel for the control plane traffic between a base station process and the EPC. For testing purposes, a simple MME node process will also be developed to act as initial S1AP communication partner with the base station process. This MME node process is intended to provide only very basic S1AP related functionality, but additional features can be developed in the future.

Due to the complexity of the full feature set of S1AP, only basic functionality of S1AP is implemented in this work. However, the protocol itself is fully implemented allowing any needed functionality to be added in the future with minimal knowledge of the internal S1AP implementation details. Only understanding of the functional details, and the data contained inside S1AP messages, will be needed. The S1AP implementation hides all the internal details behind an easy to use interface.

To start, chapter 2 introduces background information on the implementation environment of this work. This includes the basic architecture of LTE networks and how the S1 interface is related to it. Relevant details of the S1 interface are also explained. Next, the current functionality and structure of the Aalto Radio Framework (ARF) platform is presented. Chapter 3 acts as an introduction and quick tutorial on Abstract Syntax Notation One (ASN.1). This is the syntax used to specify the S1AP. Basic understanding of ASN.1 is required to use and understand S1AP. After this, chapter 4 dives into the details of the S1AP specifications and ASN.1 is used in them. The chapter also explains in detail how the ASN syntax of S1AP is used together with the open source ASN.1 compiler asn1c to generate a working implementation of S1AP. The limitations of asn1c are discussed and methods to work around them are introduced. The following chapter 5 explains how the S1AP implementation is integrated into the ARF platform and how an application programming interface (API) is built to provide easy access to the S1AP implementation. The chapter also explains how the implementation was tested. Chapter 6 provides the summary of the work and a discussion on future implementation and improvement goals for ARF.

2 Implementation environment

To properly implement the S1 Application Protocol, the complete environment where it is to be used must first be understood. In this chapter, this environment is introduced and explained. First, the LTE network architecture is presented in a general level, explaining the place and role of S1AP in it. Next, the S1 interface between the E-UTRAN and the EPC is introduced in more detail. Lastly, the Aalto Radio Framework (ARF) software is introduced and the basic working principles behind it are explained.

2.1 LTE network architecture

An LTE network is formed mainly by two parts, the air interface and the core network. The air interface is called Evolved Universal Terrestrial Radio Access Network (E-UTRAN) and it consists only of E-UTRAN Node Bs (eNodeB) on the network side. These are the base stations in an LTE network. The core network architecture of LTE is called System Architecture Evolution (SAE). It has a packet switched, all-IP architecture and all control plane and user plane traffic is separated [1]. The main component of the SAE architecture is the Evolved Packet Core (EPC). Figure 1 illustrates this LTE network architecture. It also shows the main nodes of the EPC and the different interfaces betweed them, which will be discussed next.

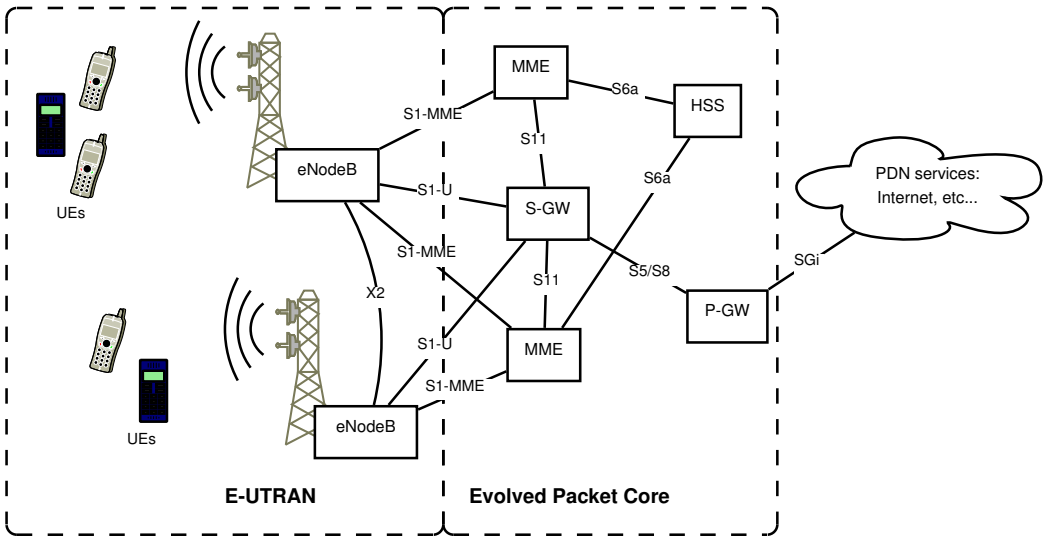

Figure 1: LTE network architecture.

The main subcomponents of the EPC are the Mobility Management Entity (MME), the Serving Gateway (S-GW), the PDN Gateway (P-GW) and the Home Subscriber Server (HSS). The MME is the key control-node for the LTE access-network. These nodes are in charge of all the control plane functions related to subscriber and session management. This includes support for e. g. terminal-to-network session handling, idle terminal location management and security procedures. Terminal-to-network session handling relates to all the signalling procedures used to

set up connections and negotiate associated parameters like Quality of Service (QoS). Idle terminal location management relates to the tracking area (TA) update process used to enable the network to join idle terminals in case of incoming connections. Security procedures relate to end-user authentication as well as the initiation and negotiation of ciphering and integrity protection algorithms. The Serving Gateway routes and forwards user data packets. It is the termination point of the packet data interface towards E-UTRAN. The PDN Gateway is the termination point of the packet data interface towards Packet Data Networks (PDN). The P-GW performs e. g. policy enforcement features, user specific packet filtering, charging support and lawful interception. Another key role of the P-GW is to act as the anchor for mobility between 3GPP and non-3GPP technologies. Lastly, the Home Subscriber Server is the combination of Home Location Register (HLR) and Authentication Center (AuC). It is a central database that contains user-related and subscription-related information. The HLR part is in charge of storing and updating the database containing all the user subscription related information. The AuC part is in charge of generating security information from user identity keys. This security information is provided to the HLR and further communicated to other entities in the network. The MME is connected to the HSS through the S6 interface, and to the S-GW through the S11 interface. The S-GW is connected to the P-GW through the S5/S8 interface.

The E-UTRAN uses a simplified single node architecture consisting only of eNodeBs. The eNodeBs can communicate with other eNodeBs through the X2 interface. This interface is split into X2-C and X2-U for control and user plane respectively. The eNodeB communicates with the EPC using the S1 interface, specifically with the MME and the User Plane Entity (UPE) identified as S-GW using the S1-MME and S1-U for control plane and user plane respectively. The MME and S-GW are logical nodes in the EPC, so they can be implemented as a single network node. This is however not recommended. By implementing them as separate network nodes instead, independent scaling of the control and user plane becomes significantly easier [1].

2.2 The S1 interface

As described in the previous section, the S1 interface links the E-UTRAN and EPC together. The interface is separated into the control plane and user plane. From the S1 perspective, the E-UTRAN access point is an eNodeB, and the EPC access point is either the control plane MME logical node or the user plane S-GW logical node. Two types of S1 interfaces are thus defined at the boundary depending on the EPC access point. These are S1-MME towards an MME and S1-U towards an S-GW [2].

An eNodeB can be connected to multiple MMEs and S-GWs. In such cases, a NAS node selection function in the eNodeB will determine and establish an association between a given User Equipment (UE) and one of the MME nodes that comprise the pool area the eNodeB belongs to. The S1-U interface selection is however done within the EPC and signalled to the eNodeB by the MME.

Figure 2 depicts the protocol stack for both the S1-MME and the S1-U interfaces. For the S1-U interface, any data link layer that fulfils the requirements toward the

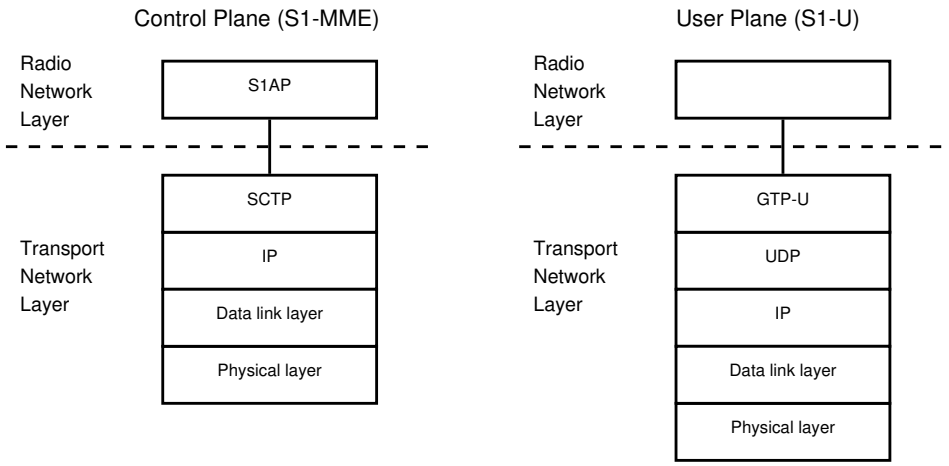

Figure 2: Interface protocol structure for S1-MME (control plane) and S1-U (user plane).

upper layer can be used. The transport for data streams is done with the GTP-U protocol over UDP over IP, and the IP layer must support IPv4 and/or IPv6 [6].

The S1-MME interface is the signalling transport to be used across the S1 interface, and the S1AP signalling messages are transported over it. First, any suitable data link layer protocol, like Ethernet or PPP, is supported [3]. Next, the IP layer must support IPv4 and/or IPv6, and only point-to-point transmission for delivering S1AP messages is supported. On top of the IP layer, Stream Control Transmission Protocol (SCTP) is used as the transport layer [4].

SCTP was originally designed to transport Public Switched Telephone Network (PSTN) signalling messages over IP networks [24]. It is a reliable transport protocol operating on top of a connectionless packet network. As it's main features, SCTP provides acknowledged error-free non-duplicated transfer of user data and sequenced delivery of user messages within multiple streams. SCTP also provides network-level fault tolerance through multi-homing support, resistance to flooding and masquerade attacks, and congestion avoidance behaviour. SCTP was specifically designed to bypass signalling protocol relevant limitations in TCP. The main limitation being the delay caused by the strict order-of-transmission delivery of data in TCP.

Between one eNodeB and MME pair, there will be only one SCTP association, and this is established by the eNodeB. The payload protocol identifier assigned by IANA to be used by SCTP for the application layer protocol S1AP is 18, and the SCTP destination port number value assigned by IANA for S1AP is 36412 [2]. SCTP multi-homing between the two end-points, where one or both are assigned with multiple IP addresses, may be used to improve transport network redundancy. Within the established SCTP association between an eNodeB and an MME, a single stream is reserved for the sole use of S1AP messages used in non UE-associated signalling. At least one stream must be reserved for S1AP messages used in UE-associated signalling, but two or more is recommended by the specifications. Also, the stream used for signalling messages related to a specific UE should not change during the lifetime of the connection context.

When a new UE-associated logical connection is created in either an eNodeB or an MME, an Application Protocol Identity (AP ID) is allocated for it. This AP ID will uniquely identify the logical connection over the S1 and X2 interfaces. To be more specific, both nodes assign their own identification number to the logical connection. The MME assigns a "MME UE S1AP ID" identity and the eNodeB assigns a "eNB UE S1AP ID" identity. When receiving a new message that has a new AP ID from the sending node, the receiving node will store the AP ID of the sending node for the duration of the logical connection. Next, the receiving node will assign a new AP ID to identify the logical connection and include both AP IDs in the first returned message to the sending node. Both AP IDs will also be included in all subsequent messages to and from the sending node. An eNodeB will also store other information associated to each active UE. Such a block of information forms an eNodeB UE context, and it contains necessary information required to maintain the E-UTRAN services towards the active UE. In addition to the AP IDs, this includes security information, UE state information and UE capability information.

2.3 Aalto Radio Framework

Traditionally commercial base stations are implemented using proprietary software running on custom hardware using integrated circuits (IC). While custom hardware provides speed and high reliability regarding execution time, both of which are extremely important due to the real-time nature of baseband processing, custom hardware is difficult and expensive to modify. A purely software based solution instead would be inherently slower, but would provide much more flexibility in exchange. Any changes would be easier to implement and thus cheaper to accomplish in software, and if that software can also be run on general purpose processors (GPP), the cost of required hardware would also go down significantly. Cloud radio access networks (C-RAN) are formed using servers connected to Remote Radio Heads (RRH). If these servers run on common purpose hardware with a non real-time operating system instead of specialized hardware, significantly reduced network operation and maintenance costs can be achieved. To achieve sufficient capacity for the radio access network, the software running on these servers needs to also be designed to support parallelism. This way capacity can be increased in the network simply by adding more servers instead of requiring more speed from existing servers. This in turn enables the use of whatever happens to be the most cost-effective general purpose hardware at a give time, leading to further cost reductions. Baseband processing in a radio access networks is a highly time critical process. To be able to adequately handle this process is the biggest challenge in C-RAN implementation. The solution involves solving two related problems. First, the amount of any timing related errors must be minimized. Due to using general purpose processors, these errors can not be completely eliminated. Second, the errors that do happen due to missed timing constraints, must be handled in a way that still allows the network to operate.

The Aalto Radio Framework (ARF) is a software framework developed in C++ that originally implements part of the time-division duplexing (TDD) LTE Release 8 specification. The main goal of the ARF software is to be a flexible research tool for

the study of radio technologies. It is useful for testing new radio features, building proof-of-concepts and for use in teaching and learning purposes. The ARF software is designed to run on generic PC hardware running the Linux operating system. The development work has been carried out using versions 14.04 LTS and 16.04 LTS of the the Ubuntu Linux distribution. It is not designed for any commercial levels of reliability, but is sufficient enough to produce valid research and scientific measurement data. A radio front-end (RF-front-end) is also required to be able to transmit and receive the actual radio signals. The RF-front-end performs the digital-to-analogue (DAC) and analogue-to-digital (ADC) conversions, modulation of the carrier frequency and transmission and reception of radio frequency (RF) signals. One good option for an RF-front-end would be the Universal Software Radio Peripheral (USRP) product family, which is a range of software-defined radios (SDR) designed and sold by Ettus Research. This product family is a comparatively inexpensive hardware platform for software radio commonly used by research labs and universities.

The ARF software implementation can be divided into two main parts, the infrastructure implementation for the framework, and the protocol specific implementation. The software is structured in such a way that the infrastructure runs the protocol specific implementation, but they are not dependent on each other. This enables the possibility to replace the protocol specific implementation with another similar radio protocol. The architecture of the protocol specific implementation is also very modular, enabling the option of replacing only specific modules while keeping the rest of the implementation untouched. This facilitates and speeds up the implementation work for any desired changes.

The implementation of the ARF software is split into three levels, the higher layer logic level, the pipes level and the hardware layer level [18]. From a timing perspective, the levels operate independently from each other, maintaining the independent modular architecture. Each level is separated by an interface containing protocol independent timestamped buffers, which manage the communication between those levels. Figure 3 illustrates this architecture. In this figure, each component is also coloured according to the category it belongs to. The blue components are part of the infrastructure implementation, including all the support functions. For clarity, the support functions are not connected to any specific components, but they are available in most parts of the framework as required. As the name implies, they provide generic supporting functionality, for example logging, debugging support and statistics gathering. The red components represent the protocol specific parts of the implementation. These would need to be replaced or modified to implement another protocol. Finally, the green component represents the interface between the radio interface implementation and the rest of the application.

Each interface between the different levels, as depicted in figure 3, is implemented in a protocol independent manner. This enables the use of multiple protocols at the same time. The higher layer logic level is built around the message router. Depending on the number of protocols, there can be several message routers. Each protocol having their own specific implementation. The different internal modules of a protocol act as message agents and all communication goes through the message

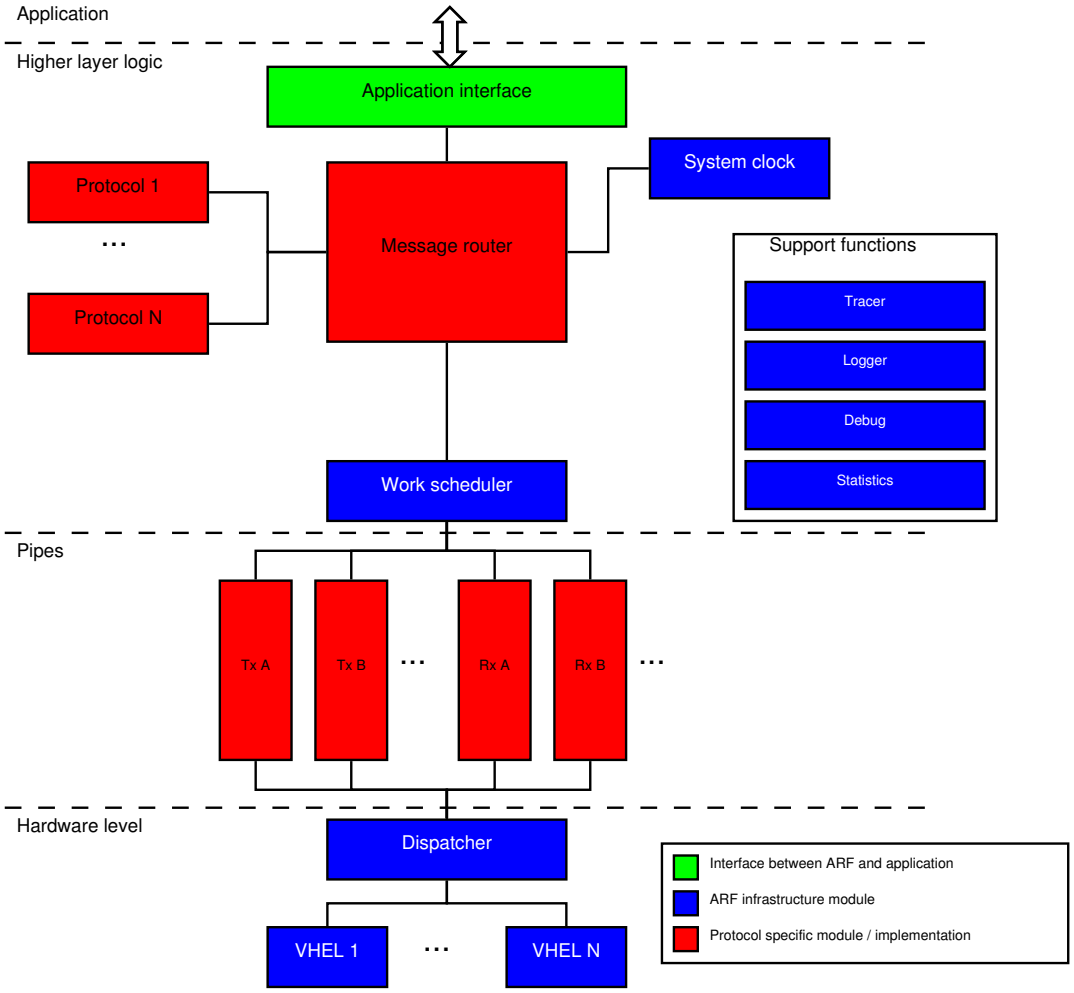

Figure 3: ARF architecture overview.

routers. For example, the protocol stack for the air interface of LTE consists of three layers, the physical (PHY) layer (TS-36.211), the medium access control (MAC) layer (TS-36.321) and the radio link control (RLC) layer (TS-36.322). Each layer has its own implementation in ARF and they communicate using the LTE specific message router implementation. All protocol related state information is kept in the the higher layer logic level. This simplifies the implementation of the lower levels and allows parallelism. The protocol implementation produces independent work items that are passed to the pipes for processing. In the case of LTE, each work item contains one subframe, which equates to 1 ms worth of data.

The pipes level consists of two types of protocol specific pipes. One type is for transmitting data (Tx) and the other for receiving data (Rx). Each pipe is an independent processing unit that contains no state information. They are only running when they are given work to do, for Tx pipes from the upper level work scheduler or for Rx pipes from the lower level dispatcher. Depending on the required capacity, there can be different amounts of Tx and Rx pipes. For optimal performance, each pipe should be run on their own processor core to allow fast execution. The

performance of the pipes is the main factor in achieving the strict timing requirements of platform.

The hardware level consists of a number of Virtual Hardware Enhancement Layer (VHEL) objects and a dispatcher that coordinates the data streams between them and the pipes. The role of a VHEL is to provide or receive a steady stream of samples from the RF-front-end. This requires converting between logical subframe level timing used in the radio protocol logic to the sample level timing used by the RF-front-end. This requires that the processed subframes are delivered on time from the pipes. However, sometimes this is not possible and some subframes inevitably arrive late. In such cases, the VHEL will replace the missing data by zeroes and thus allow the system to continue operating normally. The resulting errors appear as channel errors to the receiver, which in practice just lowers the achieved throughput slightly [17].

The ARF implements the air interface of an eNodeB, an LTE base station. To become a full eNodeB implementation, it would also require the implementation of the X2 interface to communicate with other eNodeBs, and the S1 interface to communicate with the LTE network core. In addition, all the logic required to manage these interfaces also still needs to be implemented. The control plane of the S1 interface uses the S1 Application Protocol (S1AP). Implementing it is a logical and required step towards a full eNodeB implementation. The next chapter will introduce ASN.1, which is the syntax used to define the S1AP.

3 ASN.1

Abstract Syntax Notation One (ASN.1) is a standard and notation designed to formally describe the semantics of data, which is intended to be transmitted across networks between heterogeneous systems. It is a notation much like any programming language, but is used to describe data structures. It has a specific syntax and a clearly defined set of rules it conforms to [7].

ASN.1 is introduced here because it has been used in the S1AP specifications. In the S1AP, each message and each data structure within those messages is defined with ASN.1. The S1AP specifications contain approximately a hundred pages of ASN.1 syntax. To create an implementation of this protocol, it is necessary to fully understand this syntax.

The main purpose of ASN.1 is to let protocol designers focus on the content of their protocol messages by allowing them to ignore the bits and bytes layout of those messages. It provides a high level description of messages that frees protocol designers from considering the actual bit layout of each message. Instead of thinking how some data should be stored in a message, a protocol designer can instead spend more of his time thinking about what data should be included in a message. The bit layout of protocol messages, as they are transferred between communicating application programs, is defined by a set of encoding rules that supplement the ASN.1 notation.

ASN.1 contains a large set of different encoding rules. These encoding rules define how data structures should be converted into bit streams and also how they can be decoded back into the original form data structures. Some of the available encoding rules are for example the Basic Encoding Rules (BER) [11], the Packed Encoding Rules (PER) [12], the XML Encoding Rules (XER) [13] and the Octet Encoding Rules [14]. There are many other encoding rules too, and new ones can always be developed. A few encoding rules have also been developed to take advantage of the internal data structure of a specific protocol, and thus work particularly well with that protocol. Different encoding rules have different properties and design goals. For example, BER is very simple, but uses a lot of bits. On the other hand, PER is significantly more complex, but manages to squeeze information into a much smaller number of bits.

ASN.1 also provides very powerful mechanisms for defining extensible data structures. These are data structures that can be updated and modified later in such a way, that they still remain backwards compatible with older versions. This is often a critical feature in protocol design, for it allows protocols to be fixed and extended in the future if needed.

Neither the abstract syntax nor its encoding rules are tied to any specific language, operating system or computer architecture, and are used in a wide range of programming languages, including for example C++, C, Java and C-sharp.

3.1 History

The formal standards documents on ASN.1 and its encoding rules are jointly published by the International Telecommunication Union-Telecommunication Standardization

Sector (ITU-T), International Organization for Standardization (ISO), and International Electrotechnical Commission (IEC). Originally, ASN.1 was defined already in 1984 as part of CCITT X.409. It was moved to it's own standard, X.208, in 1988 due to wide applicability. In 1995, a substantially revised version was published as the X.680 series [7]. This version did some major changes to the standard, like replacing macros with information object classes. The latest revision of the X.680 series of recommendations is the 5.0 edition, published in august, 2015.

3.2 Syntax and semantics

The purpose of ASN.1 is to describe data structures. The main concept to do this is the type. A type is a non-empty set of values, which can be encoded to be transmitted. ASN.1 provides a set of basic types, like INTEGER, BOOLEAN, ENUMERATED and BIT STRING for example. These are introduced in greater detail later in this chapter. ASN.1 also provides a set of constructed types, like SEQUENCE, SET and CHOICE. These are also introduced in greater detail later in this chapter. More complex types can be defined by combining the basic types with the constructed types. When defining a new more complex type, it is given a name that must start with a capital letter and be unique within the specification. This new type can then be used, just like any of the basic types, to further define even more new types. In this manner, a single type can represent a complex tree-like data structure.

3.2.1 Assignments

There are six distinct categories of assignments: types, values, value sets, information object classes, information objects and information object sets.

A new type is defined by first giving it a name starting with an upper case letter, followed by the symbol "::=" and a type expression. A type expression is some combination of basic types, custom types and constructed types. These are all explained in later parts of this chapter.

```
TypeReference ::= CHOICE {
    name-of-the-integer INTEGER,
    name-of-the-boolean BOOLEAN
}
```

A value is defined by giving it a name starting with a lower case letter, followed by its type, the "::=" symbol and the value itself. The type is normally referenced by using its name, which starts with an upper case letter. Optionally, the type name could be replaced by its definition, but this makes the syntax less clear and harder to read.

```
value-reference TypeReference ::= value
```

A value set is defined by giving it a name starting with an upper case letter, followed by its type, the "::=" symbol and last a series of values in curly brackets separated by the vertical bar "|".

```
GfxModelNumber INTEGER ::= { 960 | 970 | 980 | 1070 | 1080 }
```

An information object class is a grouping of elements that share some common characteristic. The elements can be types, values, value sets, information objects and information object sets. The name of an information object class must be all in upper case. In order to avoid confusion with other pieces of ASN.1 notation, the names of fields of Information Object Classes are required to begin with an ampersand character ('&').

```
FUNCTION ::= CLASS {
    &ArgumentType,
    &ResultType     DEFAULT NULL,
    &Errors         ERROR OPTIONAL,
    &opcode         INTEGER UNIQUE
}
```

An information object is an instance of an information object class. Names given to individual information objects are required to start with a lower case letter, similar to value references.

An information object set is a collection of information objects of a given class, just like a value set is a collection of values of a certain type. Names given to information object sets are required to start with an upper case letter, similar to a value set.

3.2.2 Basic types

This section introduces the ASN.1 basic types that are relevant to the understanding of the S1AP specifications. In addition to the basic types that follow, ASN.1 also has a large number of character string types. These character string types differ from each other mostly by what character alphabets they support and what types of encodings they use. Character string types are rarely used in protocol specifications because data in string format is rarely needed and usually their use only adds unnecessary complexity. Thus character strings are not addressed in this document. More information can be found from the ASN.1 specifications if needed [7].

BOOLEAN

Simplest of the ASN.1 basic types, the boolean type is declared with the keyword BOOLEAN. A boolean type can have the value of TRUE or FALSE. In the case where a group of boolean values is modelled, a BIT STRING type might be used instead, with each bit referring to a separate boolean value. This type is introduced later.

```
AnswerResult ::= BOOLEAN
right AnswerResult ::= TRUE
wrong AnswerResult ::= FALSE
```

NULL

The null type has only one possible value, which is the same as the declaration keyword: NULL. The type itself carries no information, but the presence of a null type does. For example, a null type could be used to model an acknowledgement or perhaps and error state inside a CHOICE type. Example:

```
Ack ::= NULL
Sensor ::= CHOICE {
    measurement-value  INTEGER,
    out-of-order       NULL
}
```

INTEGER

The keyword INTEGER declares a positive or negative integer of any length, including zero. Since it is much harder and more complex to write applications that can properly handle integers of unlimited size, it is generally good practice to write ASN.1 specifications where value range constraints are added on INTEGER types whenever possible [9]. Example:

```
minus-three INTEGER ::= -3
DayOfMonth ::= INTEGER (1..31)
StratumLevel ::= INTEGER (0..3, ...)
```

In the example above, the first line declares a value reference 'minus-three' that is given the negative value minus three. Note that the name 'minus-three' must start with a lower case letter. The second line defines a new type 'DayOfMonth', which is a subtype of INTEGER, constrained to values greater than zero and less than 32. Since this is a type definition, the name starts with an upper case letter. The third line defines a similar INTEGER subtype, but this time the value constraint is extensible, and could thus be changed in a future version of the specification [9].

ENUMERATED

The enumerations type is declared with the keyword ENUMERATED, and it is used to create a list of identifiers. These identifiers are named integers, but they are not numbers and thus cannot be manipulated by operators. The identifiers can be explicitly associated with unique integer values, or they can be assigned automatically according to the rules specified in the ASN.1 standard. In any case, those values are only used in encoding. Enumerations can also be made extensible by using the extension marker '...'. Example:

```
Cell-Size ::= ENUMERATED {verysmall, small, medium, large, ...}
```

REAL

Real numbers can be modelled with the real type, declared with the keyword REAL. These are arbitrarily long but finite decimals. The value notation consists of a comma-separated list of three integer numbers, which are the mantissa, base and exponent. The mathematical value being identified by (mantissa times (base to the power of exponent)). The base is only allowed values of 2 or 10. Example:

```
pi REAL ::= {mantissa 314159, base 10, exponent -5}
real01 REAL ::= {mantissa 1, base 2, exponent -1}
```

BIT STRING

A sequence of bits can be represented with the bit string type. This type is declared with the keyword 'BIT STRING', the space being a mandatory part. By default, a bit string can be of zero length, or arbitrarily long. For this reason, any proper specification should always define at least a maximum length for any bit string when possible. The value notation consists of the value in quotes followed by the capital letter B if the value is presented with binary digits (0 or 1) or the capital letter H if the value is presented in hexadecimal digits (0–9 and A–F). Example:

```
value-a BIT STRING ::= '011010'B
value-b BIT STRING ::= '1A'H
SecurityKey ::= BIT STRING (SIZE(256))
```

OCTET STRING

The octet string type is identical to the bit string type, except the length is always a multiple of 8 bits. It is declared with the 'OCTET STRING' keyword. Example:

```
value-c OCTET STRING ::= '00011010'B
value-d OCTET STRING ::= '1A'H
TBCD-STRING ::= OCTET STRING (SIZE(3))
```

3.2.3 Constructed types

This section introduces the ASN.1 constructed types. These can be used to construct more complex types by combining already defined or existing types in various ways.

SEQUENCE

The sequence type provides an ordered series of elements of different type. It is declared with the keyword SEQUENCE. This construct might seem equivalent to the C-language struct construct, but it differs from it by allowing components of the sequence to be marked as optional with the use of the keyword OPTIONAL. Components may also be given default values by using the keyword DEFAULT followed by a compatible value.

```
Person ::= SEQUENCE { name IA5String,
                      age  INTEGER  OPTIONAL}
johnny Person ::= { name "John Smith",
                    age  33 }
EmergencyAreaID-Broadcast-Item ::= SEQUENCE {
    emergencyAreaID     EmergencyAreaID,
    completedCellinEAI  CompletedCellinEAI,
    ...
}
```

SEQUENCE OF

The sequence of type is equivalent to dynamic arrays found in many programming languages. All the elements in the array are of the same type, but the number of elements is not known before hand. The type is declared with the keyword 'SEQUENCE OF'. By default, the structure can have an unlimited number of elements. This can be restricted with the SIZE constraint.

```
PhoneBook ::= SEQUENCE OF PhoneRecord
BPLMNs ::= SEQUENCE (1..maxnoofBPLMNs) OF PLMNidentity
```

SET

The set type is identical to the sequence type except that the order of components does not matter in a set structure. This type is declared with the keyword SET. As a general rule, it is recommended to use the sequence type instead of the set type, because encoding and decoding ordered data is much faster [15].

```
Person ::= SET { name IA5String,
                 age  INTEGER  OPTIONAL}
```

SET OF

Similarly to the set type, the set of type is equivalent to the sequence of type, except that the order of the components does not matter. This type is declared with the keyword 'SET OF'. Also, for the same reason as for the set type, it is recommended to use the sequence of type in place of the set of type when ever possible.

```
LotteryDraw ::= SET OF INTEGER
draw LotteryDraw ::= {1, 2, 3, 4, 5, 6, 8}
```

CHOICE

The choice type represents a selection from a set of alternatives. It is declared with the keyword CHOICE. An important detail to note is that the choice type models two pieces of information: the chosen alternative and the value associated with this chosen alternative.

```
ENB-ID ::= CHOICE {
    macroENB-ID  BIT STRING(SIZE(20)),
    homeENB-ID   BIT STRING(SIZE(28)),
    ...
}
```

ANY

The any type can represent any valid ASN.1 type. It could be declared by the keyword ANY, but it has actually been removed from the ASN.1 standard since 1994 and its use is strongly inadvisable [15]. Conceptually, the any type is like a choice type but with an unlimited number of options.

3.2.4 Extensibility

The ability to fix unforeseen problems or to add new features in the future is an important part of protocol design. ASN.1 provides extensibility mechanisms to provide support for exactly this kind of behaviour. These extensibility mechanisms help make it possible for successive versions of the same specification to be compatible with each other. However, this is not something that just happens automatically. Extensibility needs to be 'baked-in' the specifications. This is done by adding the ASN.1 extensibility marker, the ellipsis (...), into the ASN.1 syntax at specific locations. In the future, new elements or modifications can be added into these locations. It is important to note that the rules concerning the use of the extensibility marker are precisely documented in the ASN.1 standard. This document does not go into further details concerning these rules.

3.2.5 Information Object Classes

An information object class is a grouping of elements that share some common characteristic. The regular method to define an information object, which is an instance of an information object class, would be to list the contents of the object in a 'two-column' format and enclose that list in curly brackets [8]. This list would associate a value with each field name in the class, but the required format can be quite complex as it needs to strictly follow ASN.1 syntax rules. However, if the information object class definition contains a section declared with the keywords WITH SYNTAX, an alternative syntax to declare objects of this type of class is specified. This custom syntax consists of a phrase with 'gaps'. All the words in the phrase are in capital letters and the gaps are the class field names that begin with

the ampersand (&) character. If a field in the class is marked with OPTIONAL or DEFAULT, the corresponding part of the custom syntax phrase must be written inside square brackets to indicate it is optional. When this custom syntax is used to specify an information object of this class, the phrase is simply written out inside curly brackets and each gap is filled with the intended value. When the custom syntax is specified in a proper way, this results in a more user-friendly and natural looking object specification. This in turn makes specifications easier to design [15].

3.3 Encoding rules

ASN.1 defines an abstract syntax, a method to describe the structure of information. By itself, this does not indicate in any way how that information should be represented in bits and bytes when transferred to another party via a digital communication channel. The bit patterns that are transferred through a communication channel are called the transfer syntax. Linking the transfer syntax to the abstract syntax, is the encoding rules.

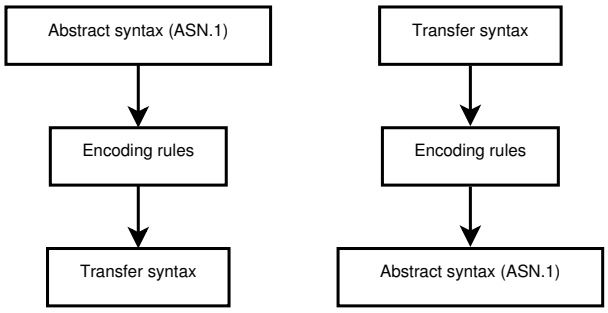

Figure 4: Encoding rules provide a conversion between abstract syntax and trasfer syntax.

Figure 4 illustrates the basic principle behind encoding rules. Applying specific encoding rules to the abstract syntax of some information results in the transfer syntax of that information. Applying the same encoding rules in reverse to the transfer syntax in turn results in the corresponding abstract syntax. Before applying encoding rules, it is assumed that the ASN.1 specification to be used is semantically correct.

There are several different encoding rules, and new ones can always be developed. Different encoding rules have different properties, like size of the generated bit patterns, complexity or suitability for a specific type of information. The encoding rules define how to represent with a bit pattern the abstract values in each basic ASN.1 type, and those in any possible constructed type that can be defined using the ASN.1 notation. This leads to the fact that previously developed ASN.1 specifications can be used with newly developed encoding rules without any modifications. The following text will briefly present the basics of the most commonly used encoding rules.

3.3.1 Basic Encoding Rules

The Basic Encoding Rules (BER) are historically the original encoding rules of ASN.1. They were already part of the standard before it was split up into two parts in 1985. To describe it shortly, BER is simple, robust but inefficient. BER also does not take into account any type constraints, since it was developed before constraints were added into the standard in 1986.

The main principle of the BER transfer syntax is the TLV triplet (Type, Length and Value). This structure, illustrated in figure 5, contains 3 parts. First is the type field T, which indicates the type of the information contained in the value field V. Next is the Length field L, which contains the length of the value field in octets. Last is the value field, that contains the actual information. If the length field contains a length of 0, then the value field is effectively missing.

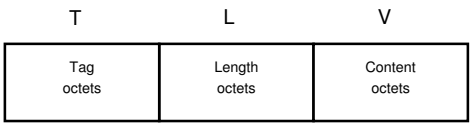

Figure 5: Triplet TLV (Type, Length, Value).

A BER encoding is a stream of TLV triplets, but a value field inside a triplet can itself contain one ore more TLV triplets, and so on. This recursive principle is demonstrated in figure 6. This means a complex ASN.1 value is no more than a stack of less and less complex values.

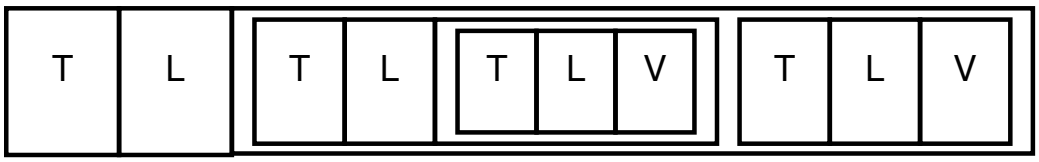

Figure 6: TLV recursive principle.

BER is octet based, so each field is a multiple of 8 bits. The type field consists of one or more octets. The first octet contains two bits of class information and one bit to indicate the form of the value field. This leaves 5 bits, which is enough to encode tag values from 0 to 30. Tag value 31 is reserved for future use. These small tag values form the UNIVERSAL tag class, and they contain all the basic ASN.1 types. This also means that the type field for these ASN.1 types only needs one octet.

Use of larger tag values is indicated by the two class bits in the first octet and they are encoded inside a required number of following octets. Each octet provides seven bits of capacity to store the tag value and one bit is used to mark the last octet of the type field. The form bit inside the first octet indicates if the value field is in primitive or constructed form. The length field consists of one or more octets. If the type field indicates the value field is in primitive encoding form, then the length field must be in definitive form. If the value field is in a constructed encoding form, the sender may choose to encode the length in definitive or in-definitive form. When

the length is in definitive form, it can be encoded in short or long form, depending on what the sender chooses. The short form can indicate a length of at most 127 octets. In long form, the first octet of the length field is used to indicate the length of the remainder of the length field, which can be at most 126 octets. In this case 127 is a reserved number. These at most 126 octets can encode a maximum length of 32255, or ((256*126)-1), octets for the value field. Even this is not enough, since ASN.1 can, in theory, be used to represent unlimited values. The in-definitive form of the length field allows an unlimited length for the value field. Instead of giving a length value up front, a predefined byte value is given as the first octet of the length field. All octets following this value belong to the value field until two specific end-of-content bytes are received. These are zero bytes, which naturally means the value field encoding must be such that it never contains two consecutive zero bytes.

3.3.2 Canonical and Distinguished Encoding Rules

One important feature to note in the BER encoding rules is the amount of sender choice contained in BER. The sender is free to choose many details when generating an encoding. The length field already contains several alternatives. Nothing would stop a sender to always encode the length field in the definitive long form, for example. However, even the different types of value fields can contain sender choice. As an example, the ASN.1 basic type BOOLEAN is encoded using one octet in the value field. The value false is indicated by zero, and the value true is indicated by any non-zero value. This text will not go into more details on how the different type value fields are encoded. These details can be best studied by looking at the encoding rule standards themselves.

The presence of sender choice in BER started to eventually cause trouble. For example, it made compliance tests for protocols time-consuming and expensive. This led to the development of Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). They were introduced into the ASN.1 standard in 1994 [15]. Both are actually subsets of BER. Both CER and DER define more restrictive rules to BER resulting in no degrees of freedom left when generating an encoding. This also means that both are compatible with a BER decoder.

3.3.3 Packed Encoding Rules

BER encodings use on average 50% more bits that the actual data to encode, and there started to form an increasing need for a more efficient method to encode ASN.1 data [15]. BER has two features that are its main source of inefficiency. These are the inclusion of so much type information, which is already available from the used ASN.1 specifications, and the mostly inefficient encoding of actual value information. A simple example is the BOOLEAN type, where a single bit would be enough to transfer the relevant information. BER uses, at minimum, one octet for the value field and two more octets for the type and length fields. The encoding of value information can further be improved by taking into account any available constraint information in the used ASN.1 specifications. For example the following definition of an INTEGER subtype 'Threshold' has only four possible values. Only two bits

are necessary to convey any relevant information whenever a parameter of type 'Threshold' is used.

```
Threshold ::= INTEGER(8..11)
```

These issues led to the development of Packed Encoding Rules (PER), which were introduced in the standard in 1994 [16]. PER is not based in a fixed TLV structure like BER. It is also bit oriented, while BER is octet oriented. PER encodings are formed with a recursive PLV structure (Preamble, Length, Value). All three fields are bit fields and they are all optional. PER does not contain any type or tag fields. The order of information in a PER encoded bit stream is instead based completely on the used ASN.1 specifications. This lack of a systematic fixed structure in the encoding itself means that decoding any PER encoded bit stream is impossible without the ASN.1 specifications used to generate it.

The preamble field is used to provide any necessary information about the currently encoded type. For CHOICE constructs, this includes an index value to indicate the selected alternative. The alternatives are indexed in tag order. For SET and SEQUENCE constructs, if any optional types are included, a bit map is used to indicate what elements are present in the encoding. One bit for each element, set to 1 if the element is present, 0 if not. Also for SET constructs, the sender can no longer decide the order of elements. It is required instead that the elements must be transferred in tag order.

The length field is only used when necessary, that is, when the length can not be statically determined by looking at the used ASN.1 specifications. This makes full use of any size constraints to determine the required size of different types. Depending on the situation, the length field value can also have different meanings. It can indicate length in octets, as in BER. It can indicate length in bits, which is used for the length of an unconstrained BIT STRING value for example. It can also indicate the number of iterations, which is used to determine the length of a SEQUENCE OF or SET OF value.

Without an implicit type or tag field, extensibility is not as simple to provide as in BER. This was solved by introducing the extensibility marker. The marker is the ellipsis, marked as three dots '...', and it is inserted into the ASN.1 specifications whenever something should be potentially extensible in the future. As explained in section 3.2.4, the extensibility marker can only be used in subtype constraints and ASN.1 types ENUMERATED , SEQUENCE , SET and CHOICE. PER handles this extensibility marker by inserting one bit whenever an extension marker is encountered to indicate if the current type has been extended or not. The use of the extensibility marker also enables the backwards compatibility quality of ASN.1 when using PER. Using a different version of the ASN.1 specifications when decoding a PER encoded bit stream is fully supported.

Lastly, the value field contains the actual value information. This can be the encoded value of a basic ASN.1 type, or another PLV structure. Information objects and information object sets are never encoded. Only the information itself that they contain is encoded. The structure and format provided by information object

classes is already available from the used ASN.1 specification. The specific encoding methods for the different ASN.1 types can be found from the PER standard [12].

PER actually contains four variations. The normal widely used PER variant is also called BASIC-PER. The standard also defines a more restricted version of this, called CANONICAL-PER. Although BASIC-PER already drastically reduces sender's options compared to BER, it does not remove them completely. CANONICAL-PER was defined because of this, and it restricts BASIC-PER rules in a few places so that any remaining sender's options are removed. This results in a unique canonical encoding of any ASN.1 specification. Since CANONICAL-PER is a subset of BASIC-PER, it follows that any CANONICAL-PER conformant encoding is also conformant to BASIC-PER decoding.

Depending on hardware implementation details, it might not be desirable to be constantly handling arbitrarily long bit strings that are not multiples of eight. Performance gains might be achieved if all data can be processed in octet aligned chunks. PER is by design bit oriented, which is called the UNALIGNED variant. An octet aligned version, the ALIGNED variant, was also specified, where padding bits are inserted from time to time to restore octet alignment. In the UNALIGNED variant, no padding bits are ever inserted. These variants are called ALIGNED-PER (APER) and UNALIGNED-PER (UPER). They can naturally be applied equivalently to both BASIC-PER and CANONICAL-PER. The S1AP specifications use aligned version of BASIC-PER. The next chapter introduces this protocol in detail.

4 S1 Application Protocol

In this chapter, the complete S1 Application Protocol (S1AP) implementation process is explained. The features and structure of S1AP are first explained in a general level. The next section goes through the ASN.1 syntax of S1AP specifications that defines the structure of S1AP messages in a detailed technical level. After this, the last section goes through the process of turning this ASN.1 syntax into usable application code by using an ASN.1 compiler. The end result will be a working implementation that can be used to construct, encode, decode and modify all types of messages defined in the S1AP specifications.

The S1AP consists of a set of elementary procedures. Each elementary procedure is a unit of interaction between an eNodeB in the Evolved Universal Terrestrial Radio Access Network (E-UTRAN) and a Mobility Management Entity (MME) in the Evolved Packet Core (EPC). An elementary procedure consists of an initiating message and possibly a response message. The presence of a response message leads to a division of the elementary procedures into two separate classes, class 1 and class 2. Class 1 contains the elementary procedures with a defined response, which can be either success or failure, or both. Correspondingly, class 2 contains the elementary procedures without a response. These elementary procedures are considered always successful. The response messages for class 1 elementary procedures are called successful outcome for a response indicating success, and unsuccessful outcome for a response indicating failure.

Table 1 presents all class 1 elementary procedures defined in S1AP specifications release 10 version 10.5.0, and table 2 similarly presents all class 2 elementary procedures from the same specification version [5]. This version was released in May 2012 and defines a total of 48 different elementary procedures. An earlier version of the S1AP specifications, release 8 version 8.7.0, released in September 2009, defined 43 different elementary procedures [19]. A much more recent specification, release 12 version 12.6.0, released in June 2015, defines already 51 different elementary procedures [20].

As can be observed, there hasn't been that many new elementary procedures added into the S1AP specifications after its initial release. This is due to the fact that ASN.1 provides such good extensibility to it's data structures. The information element content of existing elementary procedures has changed and many new data structures have been added over the years. New elementary procedures have been added into the specifications only when completely new features and functionality, separate from the existing elementary procedures, have been defined.

The version of the S1AP specification selected for initial implementation of the S1AP is not very critical. There are benefits in both using an earlier specification version and using the latest version. The ARF currently requires only release 8 support, but for future development purposes it would be beneficial to already provide support for later versions. However, starting implementation using the latest available version would increase the amount and complexity of the internal S1AP data structures. This in turn would add complexity in the initial implementation work. Using an earlier version with less complexities simplifies the implementation

Table 1: S1AP class 1 elementary procedures.

Elementary Procedure	**Initiating Message**	**Successful Outcome**	**Unsuccessful Outcome**
Handover Preparation	HANDOVER REQUIRED	HANDOVER COMMAND	HANDOVER PREPARATION FAILURE
Handover Resource Allocation	HANDOVER REQUEST	HANDOVER REQUEST ACKNOWLEDGE	HANDOVER FAILURE
Path Switch Request	PATH SWITCH REQUEST	PATH SWITCH REQUEST ACKNOWLEDGE	PATH SWITCH REQUEST FAILURE
Handover Cancellation	HANDOVER CANCEL	HANDOVER CANCEL ACKNOWLEDGE	
E-RAB Setup	E-RAB SETUP REQUEST	E-RAB SETUP RESPONSE	
E-RAB Modify	E-RAB MODIFY REQUEST	E-RAB MODIFY RESPONSE	
E-RAB Release	E-RAB RELEASE COMMAND	E-RAB RELEASE RESPONSE	
Initial Context Setup	INITIAL CONTEXT SETUP REQUEST	INITIAL CONTEXT SETUP RESPONSE	INITIAL CONTEXT SETUP FAILURE
Reset	RESET	RESET ACKNOWLEDGE	
S1 Setup	S1 SETUP REQUEST	S1 SETUP RESPONSE	S1 SETUP FAILURE
UE Context Release	UE CONTEXT RELEASE COMMAND	UE CONTEXT RELEASE COMPLETE	
UE Context Modification	UE CONTEXT MODIFICATION REQUEST	UE CONTEXT MODIFICATION RESPONSE	UE CONTEXT MODIFICATION FAILURE
eNB Configuration Update	ENB CONFIGURATION UPDATE	ENB CONFIGURATION UPDATE ACKNOWLEDGE	ENB CONFIGURATION UPDATE FAILURE
MME Configuration Update	MME CONFIGURATION UPDATE	MME CONFIGURATION UPDATE ACKNOWLEDGE	MME CONFIGURATION UPDATE FAILURE
Write-Replace Warning	WRITE-REPLACE WARNING REQUEST	WRITE-REPLACE WARNING RESPONSE	
Kill	KILL REQUEST	KILL RESPONSE	

work, which makes it easier to focus on any possible ASN.1 syntax and compiler related problems. Getting everything to work properly on an earlier specification version first enables straightforward implementation of latter versions. How this is done in practice is demonstrated in the next section.

The implementation work in this master's thesis is based on release 10 version 10.5.0 of the S1AP specifications. This specific version was chosen because it was also the latest version of the specification supported by the OpenAirInterface™(OAI) software at the time of the implementation work [21]. It also provides a good compromise between potentially supporting newer features of S1AP needed in future implementation work of ARF, and decreasing the data structure complexity compared to later versions of S1AP specifications. The synergy with the OAI software also provides potential benefits.

The S1AP ASN.1 specifications are divided in six modules by functionality. These modules are the S1AP-PDU-Descriptions-module, the S1AP-PDU-Contents-module,

Table 2: S1AP class 2 elementary procedures.

Elementary Procedure	**Message**
Handover Notification	HANDOVER NOTIFY
E-RAB Release Indication	E-RAB RELEASE INDICATION
Paging	PAGING
Initial UE Message	INITIAL UE MESSAGE
Downlink NAS Transport	DOWNLINK NAS TRANSPORT
Uplink NAS Transport	UPLINK NAS TRANSPORT
NAS non delivery indication	NAS NON DELIVERY INDICATION
Error Indication	ERROR INDICATION
UE Context Release Request	UE CONTEXT RELEASE REQUEST
Downlink S1 CDMA2000 Tunneling	DOWNLINK S1 CDMA2000 TUNNELING
Uplink S1 CDMA2000 Tunneling	UPLINK S1 CDMA2000 TUNNELING
UE Capability Info Indication	UE CAPABILITY INFO INDICATION
eNB Status Transfer	eNB STATUS TRANSFER
MME Status Transfer	MME STATUS TRANSFER
Deactivate Trace	DEACTIVATE TRACE
Trace Start	TRACE START
Trace Failure Indication	TRACE FAILURE INDICATION
Location Reporting Control	LOCATION REPORTING CONTROL
Location Reporting Failure Indication	LOCATION REPORTING FAILURE INDICATION
Location Report	LOCATION REPORT
Overload Start	OVERLOAD START
Overload Stop	OVERLOAD STOP
eNB Direct Information Transfer	eNB DIRECT INFORMATION TRANSFER
MME Direct Information Transfer	MME DIRECT INFORMATION TRANSFER
eNB Configuration Transfer	eNB CONFIGURATION TRANSFER
MME Configuration Transfer	MME CONFIGURATION TRANSFER
Cell Traffic Trace	CELL TRAFFIC TRACE
Downlink UE Associated LPPa Transport	DOWNLINK UE ASSOCIATED LPPA TRANSPORT
Uplink UE Associated LPPa Transport	UPLINK UE ASSOCIATED LPPA TRANSPORT
Downlink Non UE Associated LPPa Transport	DOWNLINK NON UE ASSOCIATED LPPA TRANSPORT
Uplink Non UE Associated LPPa Transport	UPLINK NON UE ASSOCIATED LPPA TRANSPORT

the S1AP-IEs-module, the S1AP-CommonDataTypes-module, the S1AP-Constants-module and the S1AP-Containers-module. The S1AP-PDU-Descriptions-module defines the highest layers of S1AP. These include the interface PDU definition, the information object class structure for an elementary procedure and all the elementary procedures themselves. The S1AP-PDU-Contents-module defines the actual information element content of each message in each elementary procedure. This means that for each message, the information elements contained in those messages are listed including their criticality and presence information. The S1AP-IEs-module defines the data structure of all information elements. These can be anything from very simple data types to very complex data structures. The S1AP-CommonDataTypes-module defines a small amount of attributes used in the definition of the higher S1AP layer structures. These include the definitions of criticality, presence and several unique identification numbers used to distinguish certain elements

from each other. The S1AP-Constants-module defines all constants used in the S1AP specifications. These include numerical values to all unique IDs for elementary procedures and information elements as well as maximum allowed sizes for all lists used in the specification. Finally, the S1AP-Containers-module defines a set of information object classes called containers. These are used as storage structures to hold several information elements of the same type inside a single information element.

4.1 S1AP ASN.1 syntax

In S1AP, each message forms an independent unit of information. This is called a protocol data unit (PDU). This term is widely used in other protocols as well. The PDU is defined in the S1AP specification as follows:

```
S1AP-PDU ::= CHOICE {
    initiatingMessage       InitiatingMessage,
    successfulOutcome       SuccessfulOutcome,
    unsuccessfulOutcome     UnsuccessfulOutcome,
    ...
}
```

This defines a new ASN.1 type called S1AP-PDU, which is a CHOICE construct with three fields as options for its content. All three fields are fixed type fields, and they are formed by two elements. The first element, starting with a lower case letter, gives the name of each field. The second element, starting with an upper case letter, gives the type of each field. For example, the two elements of the first line define a field with the name initiatingMessage and it is of the type InitiatingMessage. Since the type InitiatingMessage is not a native ASN.1 type, the definition for it must be contained somewhere else in the specification. Otherwise the ASN.1 syntax would not be complete and the specifications would be invalid. The CHOICE construct also contains the ellipsis extensibility marker (...), meaning new ASN.1 types could be added into the CHOICE construct in a latter version of the specification.

The custom ASN.1 types InitiatingMessage, SuccessfulOutcome and UnsuccessfulOutcome are structurally identical. The only difference between them, apart from the type name itself, is the type of the third field, which has the field name 'value'. Here is the definition of the InitiatingMessage type:

```
InitiatingMessage ::= SEQUENCE {
    procedureCode   S1AP-ELEMENTARY-PROCEDURE.&procedureCode
                    ({S1AP-ELEMENTARY-PROCEDURES}),

    criticality     S1AP-ELEMENTARY-PROCEDURE.&criticality
                    ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode}),

    value           S1AP-ELEMENTARY-PROCEDURE.&InitiatingMessage
                    ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode})
}
```

This definition indicates that the InitiatingMessage type is a SEQUENCE construct with three fields, and it has not been marked as extensible. All three fields in the SEQUENCE are defined with three elements. The first element, starting with a lower case letter, gives a name for the field. The third element is contained in parentheses, which indicates it is some sort of constraint. To fully understand the meaning of both the second and third elements, the definitions of S1AP-ELEMENTARY-PROCEDURE and S1AP-ELEMENTARY-PROCEDURES must first be known.

The definition for S1AP-ELEMENTARY-PROCEDURE is as follows:

```
S1AP-ELEMENTARY-PROCEDURE ::= CLASS {
    &InitiatingMessage      ,
    &SuccessfulOutcome      OPTIONAL,
    &UnsuccessfulOutcome    OPTIONAL,
    &procedureCode          ProcedureCode   UNIQUE,
    &criticality            Criticality     DEFAULT ignore
}
WITH SYNTAX {
    INITIATING MESSAGE      &InitiatingMessage
    [SUCCESSFUL OUTCOME     &SuccessfulOutcome]
    [UNSUCCESSFUL OUTCOME   &UnsuccessfulOutcome]
    PROCEDURE CODE          &procedureCode
    [CRITICALITY            &criticality]
}
```

This defines an information object class with the name S1AP-ELEMENTARY-PROCEDURE. There are two sections contained in curly brackets. The first section is the information object class definition and the second section, which is optional, defines a custom syntax for the information object class. The custom syntax provides a more user-friendly and natural syntax to specify objects conforming to the given information object class structure.

The first section contains five entries separated by a comma, so the information object class S1AP-ELEMENTARY-PROCEDURE consists of five fields. By ASN.1 specifications, a field name in a information object class always starts with the ampersand (&) character. The & sign makes a distinction between the fields of a class and the components of a SEQUENCE or SET type. The case of the following letter and whatever does or does not follow after the field name is critical in defining the category of each field. The names of the first three fields start with an upper case letter, and are followed by neither a type nor a class. They are type fields, which means that in a S1AP-ELEMENTARY-PROCEDURE object, the fields will contain an ASN.1 type. The second and third fields also contain the reserved word OPTIONAL, which means these fields are optional and may be absent in a S1AP-ELEMENTARY-PROCEDURE object.

To avoid any fundamental misunderstandings, it is important to note that the S1AP specifications define custom ASN.1 types called InitiatingMessage, SuccessfulOutcome and UnsuccessfulOutcome. While these types have identical names, excluding the & character, compared to the fields in S1AP-ELEMENTARY-PROCEDURE

class, they are completely separate entities. For example, the type field &InitiatingMessage in a S1AP-ELEMENTARY-PROCEDURE object must contain an ASN.1 type. While this type could be the custom type InitiatingMessage, it could also be the custom type SuccessfulOutcome, or any other valid ASN.1 type.

The last two fields, the fourth and fifth elements in the S1AP-ELEMENTARY-PROCEDURE class definition, are fixed-type values. The field names start with a lower case letter, and are followed by a type. The definitions for both the ProcedureCode and Criticality types is found in the common definitions part of the S1AP specifications.

```
ProcedureCode ::= INTEGER (0..255)
Criticality ::= ENUMERATED { reject, ignore, notify }
```

The &procedureCode field is a fixed value of type ProcedureCode, which is a subtype of the ASN.1 basic type INTEGER with the value range constrained from 0 to less than 256. In addition, the &procedureCode field definition contains the reserved word UNIQUE, which means that no two S1AP-ELEMENTARY-PROCEDURE objects can have identical values in their &procedureCode fields.

The &criticality field is a fixed value of type Criticality, which is an ASN.1 ENUMERATED type. This type has three possible values, 'reject', 'ignore' and 'notify', and it cannot be extended. The definition of the &criticality field also has the reserved word DEFAULT, followed by a valid Criticality type value. This means that the field is mandatory in every S1AP-ELEMENTARY-PROCEDURE object, but it has the default value of 'ignore' in case it is missing from the object definition itself.

As the name indicates, the purpose of the S1AP-ELEMENTARY-PROCEDURE information object class is to represent the different elementary procedures in S1AP. The S1AP specifications use the custom syntax, specified after the keywords WITH SYNTAX in the S1AP-ELEMENTARY-PROCEDURE class definition, to define all the elementary procedures in S1AP as S1AP-ELEMENTARY-PROCEDURE objects. As described in the beginning of this chapter, only some elementary procedures have a successful outcome or unsuccessful outcome message. This is reflected in the S1AP-ELEMENTARY-PROCEDURE class definition by marking those fields optional. As an example, here is the definition of the Reset elementary procedure. It defines an information object named 'reset', and it is an instance of the information object class S1AP-ELEMENTARY-PROCEDURE.

```
reset S1AP-ELEMENTARY-PROCEDURE ::= {
    INITIATING MESSAGE  Reset
    SUCCESSFUL OUTCOME  ResetAcknowledge
    PROCEDURE CODE      id-Reset
    CRITICALITY         reject
}
```

As can be seen in table 1, the reset elementary procedure has the initiating message and successful outcome messages but no unsuccessful outcome message.

This is also seen in the S1AP-ELEMENTARY-PROCEDURE object definition for the reset elementary procedure. The optional part SUCCESSFUL OUTCOME of the syntax phrase is present, but the optional part UNSUCCESSFUL OUTCOME is not.

This definition generates a S1AP-ELEMENTARY-PROCEDURE object where the &InitiatingMessage field has the value of ASN.1 type Reset, the &SuccessfulOutcome field has the value of ASN.1 type ResetAcknowledge, the &UnsuccessfulOutcome field is absent, the &procedureCode field is assigned the value of 'id-Reset' and the &criticality field is assigned the value 'reject'. Again, to make the S1AP ASN.1 specifications valid, the custom ASN.1 types Reset and ResetAcknowledge must be defined elsewhere in the specifications. These two types represent the messages that are part of the Reset elementary procedure and their details will be introduced later in this chapter. The value of id-Reset must be defined somewhere, or it could not be used in the object definition. Since the &procedureCode field was a subtype of INTEGER, constrained from 0 to less that 256, id-Reset must have a compatible value. The &procedureCode field was also marked UNIQUE, so the S1AP specifications must provide a unique value reserved for the Reset elementary procedure information object. The definition of id-Reset can be found in the S1AP-Constants-module of the S1AP specifications and is as follows:

```
id-Reset    ProcedureCode ::= 14
```

The S1AP-Constants-module of the S1AP specifications contain a similar unique identification number for each elementary procedure in S1AP. If new elementary procedures are added in future versions, new numbers must be assigned for them there.

The definition of the InitiatingMessage type, shown earlier, made use of the S1AP-ELEMENTARY-PROCEDURE information object class. It also made use of something called S1AP-ELEMENTARY-PROCEDURES. The definition for it is shown here:

```
S1AP-ELEMENTARY-PROCEDURES S1AP-ELEMENTARY-PROCEDURE ::= {
    S1AP-ELEMENTARY-PROCEDURES-CLASS-1  |
    S1AP-ELEMENTARY-PROCEDURES-CLASS-2,
    ...
}
```

This is an information object set. It is a structure that contains a collection of objects of the S1AP-ELEMENTARY-PROCEDURE class. This definition actually contains only two other information object sets, which are like subsets that form the actual object set. The use of subsets like this adds more structure into the specifications, but it does not alter the resulting object set in any way. Of course, the subsets themselves could also be used elsewhere, but this is not the case in the S1AP specifications.

The information object set S1AP-ELEMENTARY-PROCEDURES-CLASS-1 is defined as follows:

```
S1AP-ELEMENTARY-PROCEDURES-CLASS-1 S1AP-ELEMENTARY-PROCEDURE ::= {
    handoverPreparation         |
    handoverResourceAllocation  |
    pathSwitchRequest           |
    e-RABSetup                  |
    e-RABModify                 |
    e-RABRelease                |
    initialContextSetup         |
    handoverCancel              |
    kill                        |
    reset                       |
    s1Setup                     |
    uEContextModification       |
    uEContextRelease            |
    eNBConfigurationUpdate      |
    mMEConfigurationUpdate      |
    writeReplaceWarning,
    ...
}
```

This set contains an entry for each class 1 elementary procedure in the S1AP specifications and is also extensible. Each entry in this list, that starts with a lower case letter, must be a S1AP-ELEMENTARY-PROCEDURE object, and the definition of that object must be included somewhere in the S1AP specifications. The definition of the reset S1AP-ELEMENTARY-PROCEDURE object was introduced earlier. Similar definitions exist for every other entry on this list.

The information object set S1AP-ELEMENTARY-PROCEDURES-CLASS-2 is defined in a similar manner. To save space, some of the S1AP-ELEMENTARY-PROCEDURE object names have been removed and replaced by five consecutive dots. A list of all class 2 elementary procedures can be seen in table 2.

```
S1AP-ELEMENTARY-PROCEDURES-CLASS-2 S1AP-ELEMENTARY-PROCEDURE ::= {
    handoverNotification                    |
    e-RABReleaseIndication                  |
    paging                                  |
    .....
    eNBConfigurationTransfer                |
    mMEConfigurationTransfer                |
    privateMessage,
    ...,
    downlinkUEAssociatedLPPaTransport       |
    uplinkUEAssociatedLPPaTransport         |
    downlinkNonUEAssociatedLPPaTransport    |
    uplinkNonUEAssociatedLPPaTransport
}
```

This set contains an entry for each class 2 elementary procedure in the S1AP specifications. The set is extensible and as can be seen, there are four entries that are located after the extensibility marker. This means the set has already been extended from the original version, and the new entries are those found after the extensibility marker.

The information object set S1AP-ELEMENTARY-PROCEDURES contains a S1AP-ELEMENTARY-PROCEDURE object for each elementary procedure in the S1AP specifications. This information, combined with the definition of the S1AP-ELEMENTARY-PROCEDURE information object class, can be used to fully understand the definitions of the custom ASN.1 types InitiatingMessage, SuccessfulOutcome and UnsuccessfulOutcome. For clarity, here is the definition of the InitiatingMessage type again:

```
InitiatingMessage ::= SEQUENCE {
    procedureCode  S1AP-ELEMENTARY-PROCEDURE.&procedureCode
                   ({S1AP-ELEMENTARY-PROCEDURES}),

    criticality    S1AP-ELEMENTARY-PROCEDURE.&criticality
                   ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode}),

    value          S1AP-ELEMENTARY-PROCEDURE.&InitiatingMessage
                   ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode})
}
```

This is a sequence of three fields. The first field is named 'procedureCode', and its type comes from the &procedureCode field in the S1AP-ELEMENTARY-PROCEDURE class. This type is the custom ASN.1 type ProcedureCode, which was a subtype of the INTEGER type. The first field is also restricted by the information object set S1AP-ELEMENTARY-PROCEDURES. This means that the procedureCode field can only receive a value that is already defined in an existing S1AP-ELEMENTARY-PROCEDURE object.

The second field is named 'criticality', and its type comes from the &criticality field in the S1AP-ELEMENTARY-PROCEDURE class. This type is the Criticality type, which is an ENUMERATED type. This field is similarly restricted by the information object set S1AP-ELEMENTARY-PROCEDURES. However, it is also further restricted by the term '@procedureCode'. This is called a component relation constraint. This constraint links the field value to the first component of the sequence, the procedureCode field. In practice, this means that the criticality field can only have a value that is found in a S1AP-ELEMENTARY-PROCEDURE object that belongs to the S1AP-ELEMENTARY-PROCEDURES object set, and that object must have the same &procedureCode field value as the procedureCode field in this SEQUENCE construct. Since the &procedureCode field is defined as unique, this effectively means that the value of the criticality field depends directly on the value of the procedureCode field, and matches the relevant object definition.

The third field is named 'value', and its type also comes from the S1AP-ELEMENTARY-PROCEDURE class, from the &InitiatingMessage field. The &Ini-

tiatingMessage field is a type field, which means the value field could contain any valid ASN.1 type. However, this field has the same constraints as the second field. Since the procedureCode value will be unique, it follows that the contents of the value field will come directly from the relevant information object with the matching &procedureCode field value.

The S1AP-ELEMENTARY-PROCEDURE object named 'reset' introduced earlier in this chapter makes for a good example. Here is the reset definition again:

```
reset S1AP-ELEMENTARY-PROCEDURE ::= {
    INITIATING MESSAGE  Reset
    SUCCESSFUL OUTCOME  ResetAcknowledge
    PROCEDURE CODE      id-Reset
    CRITICALITY         reject
}
```

The reset object is part of the object set S1AP-ELEMENTARY-PROCEDURES and id-Reset is an integer type unique constant. If an instance of type InitiatingMessage is created and the procedureCode field is given the value of id-Reset, the constraints in the InitiatingMessage type definition dictate the valid values for the remaining fields. The criticality field must then have the value 'reject' and the value field must a have an instance of type Reset as its value.

In a similar manner, if an instance of type SuccessfulOutcome has the procedureCode field set to the value of id-Reset, then the criticality field must have the value 'reject' and the value field must contain an instance of type ResetAcknowledge. It is also important to note that since the reset object definition is missing the optional component UNSUCCESSFUL OUTCOME, the resulting object does not have the optional field &UnsuccessfulOutcome. This results in the fact that there can be no instance of type UnsuccessfulOutcome, where the procedureCode field has the value of id-Reset.

The ASN.1 syntax that specifies the general structure of the S1AP has now been explained. This includes the definition of the PDU, the elementary procedures and the different types of messages they contain. This still leaves the contents of those messages undefined, and that issue is looked at next.

At a general level, the different messages are collections of information elements. These information elements contain either data, or more information elements. The S1AP specifications use an information object class named S1AP-PROTOCOL-IES to define the base structure of an information element. Here is the class definition:

```
S1AP-PROTOCOL-IES ::= CLASS {
    &id             ProtocolIE-ID   UNIQUE,
    &criticality    Criticality,
    &Value,
    &presence       Presence
}
WITH SYNTAX {
    ID              &id
```

```
    CRITICALITY     &criticality
    TYPE            &Value
    PRESENCE        &presence
}
```

This class contains four fields and a custom syntax to specify objects of this class. There are three fixed-type value fields, &id, &criticality and &presence, and one type field, &Value. The Criticality type was already used in S1AP-ELEMENTARY-PROCEDURE class and was introduced earlier. The Presence and ProtocolIE-ID types are defined as follows:

```
Presence ::= ENUMERATED { optional, conditional, mandatory }
ProtocolIE-ID ::= INTEGER (0..65535)
```

As can be seen from the class definition, each information element has a unique identification number, a criticality value, presence information and some ASN.1 type to contain the actual data for the information element.

The Reset type, discussed earlier, represents the initiating message in the reset elementary procedure. It is defined as follows:

```
Reset ::= SEQUENCE {
    protocolIEs     ProtocolIE-Container{ {ResetIEs} },
    ...
}
```

This defines a SEQUENCE construct that has one field named 'protocolIEs' and is extensible. The protocolIEs field is of the type ProtocolIE-Container, but this type is parameterized with the information object set ResetIEs [10]. The object set ResetIEs is defined as follows:

```
ResetIEs S1AP-PROTOCOL-IES ::= {
    { ID    id-Cause        CRITICALITY  ignore
      TYPE  Cause           PRESENCE     mandatory } |
    { ID    id-ResetType    CRITICALITY  reject
      TYPE  ResetType       PRESENCE     mandatory },
    ...
}
```

This object set defines the actual information element content of the initiating message in the reset elementary procedure. The definition makes use of the custom syntax of information object class S1AP-PROTOCOL-IES. It can be seen that this specific message contains two information elements. Both are mandatory, and their identification number, criticality information and type is defined. The object set is also extensible, so additional information elements could be added in a future version of the specification. To understand how this object set is used, the parameterized type ProtocolIE-Container must be examined first. Here are the ASN.1 definitions related to it:

```
ProtocolIE-Container {S1AP-PROTOCOL-IES : IEsSetParam} ::=
    SEQUENCE (SIZE(0..maxProtocolIEs)) OF
    ProtocolIE-Field {{IEsSetParam}}

ProtocolIE-SingleContainer {S1AP-PROTOCOL-IES : IEsSetParam} ::=
    ProtocolIE-Field {{IEsSetParam}}

ProtocolIE-Field {S1AP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
    id              S1AP-PROTOCOL-IES.&id
                    ({IEsSetParam}),

    criticality     S1AP-PROTOCOL-IES.&criticality
                    ({IEsSetParam}{@id}),

    value           S1AP-PROTOCOL-IES.&Value
                    ({IEsSetParam}{@id})
}

maxProtocolIEs INTEGER ::= 65535
```

The ProtocolIE-Container is a SEQUENCE OF construct containing a number of ProtocolIE-Field type elements. The sequence size is constrained from 0 to less than 65536, and the ProtocolIE-Field element is parameterized to carry over the original parameter to the ProtocolIE-Container. The parameter is of the information object class S1AP-PROTOCOL-IES type, meaning it can be a specific S1AP-PROTOCOL-IES object, or a set of S1AP-PROTOCOL-IES objects.

A similar type, ProtocolIE-SingleContainer, is also defined. It is identical with the ProtocolIE-Container type, except the contents are fixed to exactly one ProtocolIE-Field type element. This actually means that the ProtocolIE-SingleContainer type ends up adding nothing to the ASN.1 syntax. It is equivalent to a function that does nothing else than call another function with the exact same parameters it was given. Its only purpose is to provide consistency in the way containers are used in the S1AP ASN.1 syntax.

The ProtocolIE-Field type is a parameterized SEQUENCE construct that has 3 fields. The types of these fields come from the information object class S1AP-PROTOCOL-IES, and they are constrained by the information object set given as parameter. In the exact same way as with the earlier demonstrated InitiatingMessage type, the constraints bind valid values of the criticality and value fields to the id field. The id field can only have values found from the object set given as parameter, and the criticality and value fields will get their values from the object identified by the id field.

At this point, one can notice the absence of any presence field in the ProtocolIE-Field type. The used parameter, the object set ResetIEs, contains this presence information, but it is not a part of the actual resulting ASN.1 type. Therefore, this presence information is never transferred inside a message. There is actually no need

for this information to be transferred. It is a fixed part of the S1AP specifications, not dynamic data that can change. This demonstrates the fact that information object classes only provide structure to information, but are not an actual part of the resulting ASN.1 structures.

The ASN.1 syntax specifying how information elements are contained in a message has now been explained. This leaves only the internal structure of those information elements. As explained in the beginning of this chapter, information elements contain either data, or other information elements, which is ultimately just more data. The Reset message makes a good example of both of these cases.

As the previously introduced definition of the Reset type indicates, the information elements contained in the Reset message are specified by the ResetIEs object set. This object set contains two S1AP-PROTOCOL-IES object definitions that include the custom ASN.1 types Cause and ResetType. These types are defined as follows:

```
Cause ::= CHOICE {
    radioNetwork    CauseRadioNetwork,
    transport       CauseTransport,
    nas             CauseNas,
    protocol        CauseProtocol,
    misc            CauseMisc,
    ...
}

ResetType ::= CHOICE {
    s1-Interface        ResetAll,
    partOfS1-Interface  UE-associatedLogicalS1-ConnectionListRes,
    ...
}
```

The Cause type is an extensible CHOICE construct, and all its fields are ENUMERATION types. These definitions are not shown here. The Cause type is considered to be data, since it is formed by a tree-like structure of basic ASN.1 types. From the functional point of view, the Cause information element is used to indicate the cause for something. It is used in many S1AP messages, most commonly to indicate a cause for a request or failure. All the different causes are listed in enumeration data structures which are divided by category. The initial CHOICE construct in the Cause information element represents the choice of that category.

The ResetType type is also a CHOICE construct. It has two fields, s1-Interface and partOfS1-Interface. The types of these fields are defined as follows:

```
ResetAll ::= ENUMERATED {
    reset-all,
    ...
}

UE-associatedLogicalS1-ConnectionListRes ::=
```

```
    SEQUENCE (SIZE(1..maxNrOfIndividualS1ConnectionsToReset)) OF
        ProtocolIE-SingleContainer
            { { UE-associatedLogicalS1-ConnectionItemRes } }

maxNrOfIndividualS1ConnectionsToReset INTEGER ::= 256
```

The ResetAll type is a simple ENUMERATED type that contains only one value, 'reset-all'. The purpose of this choice and value is to indicate that the reset message is meant to affect the whole S1 interface.

The UE-associatedLogicalS1-ConnectionListRes type is more complex. This type is a container, that is, it is meant to contain other information elements. The definition indicates it is a size constrained SEQUENCE OF construct, containing a number of ProtocolIE-SingleContainer type elements. The ProtocolIE-SingleContainer type is a parameterized type and was introduced earlier. As was explained then, the ProtocolIE-SingleContainer type doesn't add anything to the ASN.1 syntax due to its definition. This can be demonstrated by replacing ProtocolIE-SingleContainer with its definition into the UE-associatedLogicalS1-ConnectionListRes type definition. The result is as follows:

```
UE-associatedLogicalS1-ConnectionListRes ::=
    SEQUENCE (SIZE(1..maxNrOfIndividualS1ConnectionsToReset)) OF
        ProtocolIE-Field
            { { UE-associatedLogicalS1-ConnectionItemRes } }
```

The meaning of this definition is exactly the same as that of the original. As an example, the same operation could be done for the ProtocolIE-Field type. This would be the result:

```
UE-associatedLogicalS1-ConnectionListRes ::=
    SEQUENCE (SIZE(1..maxNrOfIndividualS1ConnectionsToReset)) OF
        SEQUENCE {
            id           S1AP-PROTOCOL-IES.&id
                ({UE-associatedLogicalS1-ConnectionItemRes}),

            criticality  S1AP-PROTOCOL-IES.&criticality
                ({UE-associatedLogicalS1-ConnectionItemRes}{@id}),

            value        S1AP-PROTOCOL-IES.&Value
                ({UE-associatedLogicalS1-ConnectionItemRes}{@id})
        }
```

The ASN.1 syntax is not written this way in the specifications because readability would suffer considerably. To further analyse the UE-associatedLogicalS1-ConnectionListRes type, the definition of the object set UE-associatedLogicalS1-ConnectionItemRes is needed:

```
UE-associatedLogicalS1-ConnectionItemRes S1AP-PROTOCOL-IES ::= {
    { ID           id-UE-associatedLogicalS1-ConnectionItem
      CRITICALITY  reject
      TYPE         UE-associatedLogicalS1-ConnectionItem
      PRESENCE     mandatory },
    ...
}
```

This object set contains one S1AP-PROTOCOL-IES object and is extensible. The object includes the custom ASN.1 type UE-associatedLogicalS1-ConnectionItem, which is defined as follows:

```
UE-associatedLogicalS1-ConnectionItem ::= SEQUENCE {
    mME-UE-S1AP-ID  MME-UE-S1AP-ID      OPTIONAL,
    eNB-UE-S1AP-ID  ENB-UE-S1AP-ID      OPTIONAL,
    iE-Extensions   ProtocolExtensionContainer
        { { UE-associatedLogicalS1-ConnectionItemExtIEs } } OPTIONAL,
    ...
}
```

This is a SEQUENCE construct with three fields. All fields are marked optional, but the S1AP specifications actually separately specify that at least one of the identification fields, mME-UE-S1AP-ID or eNB-UE-S1AP-ID, must be present. In a situation like this, an option to mark a sequence element as conditional instead of optional could be useful, but ASN.1 does not provide such a feature. One must not confuse the ASN.1 syntax of marking elements in sequences as optional, to the Presence type used for marking information elements as mandatory, optional or conditional. The OPTIONAL keyword is an ASN.1 feature, while the Presence type is an S1AP feature.

The first two fields of UE-associatedLogicalS1-ConnectionItem are of a custom ASN.1 type. Here are their definitions:

```
MME-UE-S1AP-ID ::= INTEGER (0..4294967295)
ENB-UE-S1AP-ID ::= INTEGER (0..16777215)
```

Both fields are simple subtypes of the basic ASN.1 type INTEGER. One should note that the size constraints are chosen to fit certain byte boundaries, here 3 and 4 bytes. This helps improve the ASN.1 encoding efficiency.

The third field of UE-associatedLogicalS1-ConnectionItem is a parameterized ProtocolExtensionContainer type. The ProtocolExtensionContainer type is structurally identical to the ProtocolIE-Container type. It only uses some modified field names to keep itself clearly separate. The parameter is again an object set. Here is its definition:

```
UE-associatedLogicalS1-ConnectionItemExtIEs
    S1AP-PROTOCOL-EXTENSION ::= {
        ...
    }
```

This is an empty object set, but it is extensible. The field name, iE-Extensions, already gives a good indication to the purpose of this container field. It is meant to be a clear and confined place to add new information elements should the need arise in the future. Many S1AP information elements contain a similar iE-Extensions field, just the object set given as parameter differs. However, since the ASN.1 structures themselves are marked as extensible with the ellipsis extension marker, this gives protocol designers often two optional methods to add extensions. This leaves the designers some additional flexibility to have an influence on the structure of information elements and the data they contain.

4.2 Using an ASN.1 compiler

The previous section explained the ASN.1 syntax used in the S1AP specifications. This ASN.1 syntax is meant to be used with an ASN.1 compiler tool to generate executable code. In this work, the open source ASN.1 compiler asn1c is used [22]. This is the only free software ASN.1 compiler available that provides C/C++ code generation [23].

Unfortunately the open source ASN.1 compiler asn1c does not support all features of the ASN.1 syntax. Specifically, it provides only very basic support for information object classes, and no support for parameterization that uses information object sets. This causes problems with the S1AP specifications, since they make extensive use of both information objects and parameterization using information object sets. Using a commercially available ASN.1 compiler would provide some advantage by providing better support for these features, but by examining the above mentioned problems in more detail, it can be noticed that such features might not be fully desirable.

In the S1AP specifications, information objects and information object sets are used to provide the desired structure for S1AP messages. Each S1AP message is defined as a sequence of information elements, which is parameterized with an information element object set. This object set acts as a constraint to limit the types of information elements allowed in each message. Information elements that contain other information elements are also constrained by the use of information object sets in similar fashion.

If the used ASN.1 compiler would provide full support for information object classes and their use in parameterization, and that support would be used, the resulting code generated by the compiler would become immensely more complex. The generated code would have to implement all restrictions and constraints for all message constructs and information element container structures. Each container structure would need a different set of constraints depending on where it is being used. To implement this type of functionality, a lot of new code would need to be generated. It would also increase code complexity and this in turn would degrade the maintainability of the code. In the end, all this extra code would just be checking and enforcing the structure of different S1AP messages. While this would be a positive feature, the benefit would be quite minimal. As an alternative option, when implementing code that creates a S1AP message, the developer could just keep an eye on the S1AP specifications and make sure to follow the given message structure.

This is not a difficult task since the message structures are defined with clear tree-like structures.

To work around the limitations in the asn1c compiler, two steps can be taken. First, a part of the S1AP ASN.1 syntax can be altered to remove the use of information object classes in such a way that the base structure does not change. Second, the part of the S1AP ASN.1 syntax that can not be altered to be compatible with the asn1c compiler is instead processed by a custom script to generate code from it. The following two sections explain these two steps in more detail.

4.2.1 Modifying the S1AP ASN.1 syntax

This section describes how the S1AP ASN.1 syntax can be modified to be fully compatible with the asn1c compiler. For the asn1c compiler, the one problem in the S1AP specifications is the use of information object sets as a parameter to constrain message structures and information element containers. These constraints define what ASN.1 types are allowed to be stored in which type field. In other words, the S1AP ASN.1 syntax contains type fields that are given an information object set as parameter, indicating that the relevant type field can only be used to store specific types indicated by the information object set. The asn1c compiler does not support information objects, and thus can't parse or understand this constraint on the type fields. However, as discussed earlier, there is no need to convey this constraint information to the compiler. It is enough if the compiler just treats the relevant fields as compatible type fields.

Information object classes were originally introduced in the ASN.1 standard to replace the ANY type. As the name indicates, the ANY type could be used to store any valid ASN.1 type. This gave a bit too much freedom to protocol designers and the ANY type started to cause various problems [16]. To solve these problems, information object classes were introduced. They were designed to replace the ANY type, but in such a way that the information to be stored had to be given some structure already at the specification phase. Information object classes are a more restricted version of the ANY type, in a way. From the encoding and decoding side, nothing actually changes. Information object classes provide structure to information, but they are never encoded in the ASN.1 transfer syntax.

The asn1c compiler still fully supports the ANY type. This is why the S1AP ASN.1 syntax can be modified in such a way that the information object set parameterization is removed and the relevant type fields are changed into ANY type fields. This will result in a compatible, but less restrictive version of the ASN.1 syntax. If the information object set constraints are removed, it results in the situation that the generated code could be used to construct and encode ASN.1 structures that no longer conform to the S1AP specifications. This would, however, require a conscious effort from the developer, and the resulting invalid S1AP messages would cause decoding errors at the receiving side. Thus, such problems are easily detected and avoided. Also, when these types of constraints are removed, the resulting code generated by an ASN.1 compiler becomes much simpler and easier to understand.

As explained in the beginning of this chapter, the S1AP ASN.1 syntax is di-

vided into six modules. These are S1AP-PDU-Descriptions, S1AP-PDU-Contents, S1AP-IEs, S1AP-CommonDataTypes, S1AP-Constants and S1AP-Containers. The necessary changes in each module will be explained and demonstrated next.

The S1AP-PDU-Descriptions module only contains four ASN.1 type definitions, S1AP-PDU, InitiatingMessage, SuccessfulOutcome and UnsuccessfulOutcome. The S1AP-PDU type is compatible, but the other three must be redefined. As an example, InitiatingMessage and SuccessfulOutcome are defined as follows:

```
InitiatingMessage ::= SEQUENCE {
    procedureCode  S1AP-ELEMENTARY-PROCEDURE.&procedureCode
                   ({S1AP-ELEMENTARY-PROCEDURES}),

    criticality    S1AP-ELEMENTARY-PROCEDURE.&criticality
                   ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode}),

    value          S1AP-ELEMENTARY-PROCEDURE.&InitiatingMessage
                   ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode})
}

SuccessfulOutcome ::= SEQUENCE {
    procedureCode  S1AP-ELEMENTARY-PROCEDURE.&procedureCode
                   ({S1AP-ELEMENTARY-PROCEDURES}),

    criticality    S1AP-ELEMENTARY-PROCEDURE.&criticality
                   ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode}),

    value          S1AP-ELEMENTARY-PROCEDURE.&SuccessfulOutcome
                   ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode})
}
```

Constraints are located inside parentheses and they can just be removed. The different field types all come from the definition of the S1AP-ELEMENTARY-PROCEDURE information object class. The &procedureCode field has type ProcedureCode and the &criticality field has type Criticality. The &InitiatingMessage and &SuccessfulOutcome fields are type fields, meaning their type is determined by the information object set parameterization. These fields must be changed to the ASN.1 ANY type. The asn1c compiler compatible definitions are as follows:

```
InitiatingMessage ::= SEQUENCE {
    procedureCode   ProcedureCode,
    criticality     Criticality,
    value           ANY
}

SuccessfulOutcome ::= SEQUENCE {
    procedureCode   ProcedureCode,
```

```
    criticality     Criticality,
    value           ANY
}
```

The S1AP-Containers module contains definitions for different types of containers. These are structures for information elements that contain other information elements. There are different types of containers for the purpose of keeping different categories of information elements in separate containers. These different containers are still structurally identical to each other. The ASN.1 syntax used to define these container structures is the most complex in the S1AP specifications. The container structure for normal information elements is defined as follows:

```
ProtocolIE-Container {S1AP-PROTOCOL-IES : IEsSetParam} ::=
    SEQUENCE (SIZE (0..maxProtocolIEs)) OF
        ProtocolIE-Field {{IEsSetParam}}

ProtocolIE-SingleContainer {S1AP-PROTOCOL-IES : IEsSetParam} ::=
    ProtocolIE-Field {{IEsSetParam}}

ProtocolIE-Field {S1AP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
    id           S1AP-PROTOCOL-IES.&id           ({IEsSetParam}),
    criticality  S1AP-PROTOCOL-IES.&criticality  ({IEsSetParam}{@id}),
    value        S1AP-PROTOCOL-IES.&Value        ({IEsSetParam}{@id})
}
```

The ProtocolIE-Container and ProtocolIE-SingleContainer types are easily converted by just removing the parameterization. The ProtocolIE-Field type can be converted in a similar manner as the InitiatingMessage type was converted earlier. These definitions are transformed as follows:

```
ProtocolIE-Container ::= SEQUENCE (SIZE (0..maxProtocolIEs)) OF
    ProtocolIE-Field

ProtocolIE-SingleContainer ::= ProtocolIE-Field

ProtocolIE-Field ::= SEQUENCE {
    id           ProtocolIE-ID,
    criticality  Criticality,
    value        ANY
}
```

The S1AP-Containers module also contains a definition for the ProtocolIE-ContainerList type. This type is only used at the beginning of the S1AP-PDU-Contents module, where it is used to define the E-RAB-IE-ContainerList type. The definitions are as follows:

```
ProtocolIE-ContainerList {INTEGER : lowerBound, INTEGER : upperBound,
        S1AP-PROTOCOL-IES : IEsSetParam} ::=
    SEQUENCE (SIZE (lowerBound..upperBound)) OF
        ProtocolIE-SingleContainer {{IEsSetParam}}

E-RAB-IE-ContainerList {S1AP-PROTOCOL-IES : IEsSetParam} ::=
    ProtocolIE-ContainerList {1, maxNrOfE-RABs, {IEsSetParam}}
```

In practice, this is just an overly complex way to define an identical container structure as seen above in the ProtocolIE-Container type. After removing the object set parameterization the definition of the E-RAB-IE-ContainerList type becomes as follows:

```
E-RAB-IE-ContainerList ::= SEQUENCE (SIZE (1..maxNrOfE-RABs)) OF
    ProtocolIE-SingleContainer
```

Since the ProtocolIE-SingleContainer type is just a renamed ProtocolIE-Field type, the above definition can be further simplified as follows:

```
E-RAB-IE-ContainerList ::= SEQUENCE (SIZE (1..maxNrOfE-RABs)) OF
    ProtocolIE-Field
```

This is identical to the ProtocolIE-Container type definition, with the exception of the size constraint parameters.

The S1AP-PDU-Contents module also includes the base information element structure of each S1AP message. In addition, any message specific information elements, ASN.1 types and containers are defined here. The message base structures and all container types always use parameterization with information object sets. The parameterization needs to be removed just as above. In addition, the definitions for the information object sets must be removed from the file since the asn1c compiler will not understand them. If left in the file, the compiler would give a parsing error. An original copy of the ASN.1 syntax file is kept, of course. In the next section, this original file will be used and the information contained in these information object set definitions will be extracted and utilised. As an example, here are the original definitions for the Reset acknowledge message:

```
ResetAcknowledge ::= SEQUENCE {
    protocolIEs ProtocolIE-Container {{ResetAcknowledgeIEs}},
    ...
}

ResetAcknowledgeIEs S1AP-PROTOCOL-IES ::= {
    { ID            id-UE-associatedLogicalS1-ConnectionListResAck
      CRITICALITY   ignore
      TYPE          UE-associatedLogicalS1-ConnectionListResAck
      PRESENCE      optional } |
```

```
    { ID           id-CriticalityDiagnostics
      CRITICALITY  ignore
      TYPE         CriticalityDiagnostics
      PRESENCE     optional },
    ...
}

UE-associatedLogicalS1-ConnectionListResAck ::=
    SEQUENCE (SIZE(1..maxNrOfIndividualS1ConnectionsToReset)) OF
        ProtocolIE-SingleContainer
            {{UE-associatedLogicalS1-ConnectionItemResAck}}

UE-associatedLogicalS1-ConnectionItemResAck S1AP-PROTOCOL-IES ::= {
    { ID           id-UE-associatedLogicalS1-ConnectionItem
      CRITICALITY  ignore
      TYPE         UE-associatedLogicalS1-ConnectionItem
      PRESENCE     mandatory },
    ...
}
```

By removing parameterization and the information object set definitions, the ASN.1 syntax becomes as follows:

```
ResetAcknowledge ::= SEQUENCE {
    protocolIEs  ProtocolIE-Container,
    ...
}

UE-associatedLogicalS1-ConnectionListResAck ::=
    SEQUENCE (SIZE(1..maxNrOfIndividualS1ConnectionsToReset)) OF
        ProtocolIE-SingleContainer
```

The S1AP-IEs module contains definitions for all the common use information elements. Some of these information elements include one or more container structures. Parameterization must be removed from these along with the definitions of the used information object sets.

The last two remaining modules, S1AP-CommonDataTypes and S1AP-Constants, do not require any alterations. One small change is nonetheless recommended to compensate for a small missing feature in the asn1c compiler. The S1AP-CommonDataTypes module contains definitions for the ProcedureCode and ProtocolIE-ID types. These are value constrained integer subtypes. The S1AP-Constants module then contains a set of numeric constants of these types with unique names and values. Unfortunately the asn1c compiler does not combine this information in the code it generates, which would be quite useful. However, if the constants are first manually merged into the type definition, the asn1c compiler will then take in those values

correctly. As an example, here is the definition of the ProcedureCode type and a few of the numeric constants of that type:

```
ProcedureCode ::= INTEGER (0..255)

id-HandoverPreparation          ProcedureCode ::= 0
id-HandoverResourceAllocation   ProcedureCode ::= 1
id-HandoverNotification         ProcedureCode ::= 2
```

The definitions of numeric constants can be integrated into the ProcedureCode type definition as follows:

```
ProcedureCode ::= INTEGER {
    id-HandoverPreparation(0),
    id-HandoverResourceAllocation(1),
    id-HandoverNotification(2)
} (0..255)
```

This will result in the proper inclusion of these constant values straight into the generated code, allowing their use from the ARF implementation side.

4.2.2 Parsing information object sets

The previous section described how the ASN.1 syntax of the S1AP specifications was altered to be compatible with the asn1c compiler. This process removed all the information object sets from the input given to the asn1c compiler. In the S1AP specifications, information object sets are used for three purposes. These three purposes are to define the different messages in elementary procedures, to define the information element content in those messages and to specify the allowed content of any information element container structures. The message content of elementary procedures is an issue best handled on the implementation side, that is, the implementation layer that makes use of the asn1c compiler generated code. This is natural since only the implementation layer would have the information of what elementary procedure and which message is needed at any given point of the program execution. The next chapter will demonstrate the details of this part of the implementation. This leaves the information object sets used in parameterization to define the content of messages and information element containers. Both are critical information needed to properly handle the ASN.1 encoding and decoding of S1AP messages.

The knowledge of what type of information elements belong in each message or container is necessary when writing the implementation layer, but it is not mandatory for the asn1c compiler itself. By removing this information from the ASN.1 syntax provided to the compiler, the ability to check the correctness of the type of information elements inside a message or container is lost. However, on the implementation layer, when creating S1AP messages and inputting data into the information elements belonging to the relevant message, the correct information elements must be known

in any case. It would be impossible to generate messages with the intended content otherwise. Taking this into account, it can be seen that the added ability to type-check messages and information element containers by the compiler is not necessary. The compiler generated code just becomes simpler and more flexible. Flexibility in this case means that the compiler generated code could also be used to construct structurally incorrect messages. to construct messages However, this does not matter because any such incorrect message would have to be purposefully constructed, would fail at the decoding stage and would be exposed in any simple testing.

Each message in S1AP consists of a set of information elements. Some of these information elements are mandatory, some are optional. To specify what information elements belong to each different message type, the S1AP specifications use information object sets. These object sets are used as constraints to the actual ASN.1 types that represent each message. They have already been introduced earlier in this chapter, but as an example, here are the ASN.1 syntax definitions for the Reset Acknowledge message and the object set used to constrain it:

```
ResetAcknowledge ::= SEQUENCE {
    protocolIEs ProtocolIE-Container {{ResetAcknowledgeIEs}},
    ...
}

ResetAcknowledgeIEs S1AP-PROTOCOL-IES ::= {
    { ID           id-UE-associatedLogicalS1-ConnectionListResAck
      CRITICALITY  ignore
      TYPE         UE-associatedLogicalS1-ConnectionListResAck
      PRESENCE     optional } |
    { ID           id-CriticalityDiagnostics
      CRITICALITY  ignore
      TYPE         CriticalityDiagnostics
      PRESENCE     optional },
    ...
}
```

The ResetAcknowledgeIEs object set defines the structure of a Reset Acknowledge message. It can contain two information elements, and both are optional. From the application implementation point of view, a data structure reflecting this message structure is needed. The data structure needs to hold all the content of a message, including the information of which optional information elements are present and which are not. This presence information is easily implemented by adding a simple flag variable, an unsigned integer where each bit signifies the presence of one optional information element. The first bit can represent the first optional element, the second bit the second optional element, and so on. A structure like this, generated for the ResetAcknowledgeIEs object set, is shown below:

```
struct ResetAcknowledgeIEs
    unsigned int
```

```
        presence_bitmask
    UE-associatedLogicalS1-ConnectionListResAck
        uE-associatedLogicalS1-ConnectionListResAck
    CriticalityDiagnostics
        criticalityDiagnostics
end struct
```

The process to generate this kind of structure is quite straight forward. The name can be taken from the object set name, a presence bit mask is added, and the name and type of the rest of the variables will come from the TYPE fields. The name in each TYPE field is an ASN.1 type. Since they represent information elements, they are also custom types defined somewhere in the S1AP specifications. This means that the asn1c compiler will process them and generate respective C/C++ data structures to represent them. Thus they can be used as such as working C/C++ variable types in the implementation code.

By examining the structure of the Reset Acknowledge message, it can be observed that the UE-associatedLogicalS1-ConnectionListResAck information element is a container structure. It is an information element used to store a number of other information elements of a certain type. Here is the ASN.1 syntax for the container definition:

```
UE-associatedLogicalS1-ConnectionListResAck ::=
    SEQUENCE (SIZE(1..maxNrOfIndividualS1ConnectionsToReset)) OF
        ProtocolIE-SingleContainer
            {{UE-associatedLogicalS1-ConnectionItemResAck}}

maxNrOfIndividualS1ConnectionsToReset INTEGER ::= 256

UE-associatedLogicalS1-ConnectionItemResAck  S1AP-PROTOCOL-IES ::= {
    { ID           id-UE-associatedLogicalS1-ConnectionItem
      CRITICALITY  ignore
      TYPE         UE-associatedLogicalS1-ConnectionItem
      PRESENCE     mandatory },
    ...
}
```

This indicates that this specific container can hold between 1 and 256 instances of the UE-associatedLogicalS1-ConnectionItem type information element. All other information element containers in the S1AP specifications are defined in a similar manner, by using an object set with only one member as a constraint.

When considering the encoding and decoding of S1AP messages, the knowledge of what information elements belong into a specific message and what belong into a certain container, becomes critical. Since the asn1c compiler does not support information object sets, this is a problem that must be solved. There are two clear solutions. Either support for information object sets should be added into the asn1c

compiler, or the required functionality that would be generated from these object set specifications must be implemented with an alternative method.

Adding new functionality to the asn1c compiler would be possible, but the compiler implementation is very complex. Understanding of the asn1c compiler would require very detailed knowledge of the several underlying ASN.1 specifications. This solution would require a large implementation effort that would go beyond the scope of this work.

The alternative is much simpler, however. The required information contained in the information object set definitions is easy to extract. The used ASN.1 syntax is identical for each information object set, and the definitions are easy to find since they all use the S1AP-PROTOCOL-IES information object class. Also, the functionality that needs to be generated from this information is structurally very simple. This is demonstrated next.

Three things are needed from the information object sets. First, a data structure that contains the required information elements is needed for each message. Secondly, an encoding function is needed for each message. This encoding function will take the data structure representing the relevant message, and encode it according to the ASN.1 PER encoding rules. And thirdly, a decoding function is needed that will reverse the operation of the encoding function. These encoding and decoding functions only need to process each message by going through the proper information elements one by one and invoking code that is generated by the asn1c compiler. This functionality is described by pseudocode in algorithms 1 and 2. Algorithm 1 provides the encoding part, and algorithm 2 provides the decoding part.

Algorithm 1 is split into two functions to simplify the structure. The first function generates a new information element and is used by the second function. The second function encodes the contents of a single message. This function must be generated separately for each message. It can be constructed from the information object set describing the information element content of the relevant message. For each information element in a message, the optionality information is processed first, and each information element present in the message is encoded by calling the first function. This will result in the encoding of the contents of the message. The final encoding of the message is left for the caller because it is the simplest solution. The PDU structure of the message has a CHOICE construct that depends on the message type. The caller has the knowledge of whether the message is of type initiating message, successful outcome or unsuccessful outcome, and can construct and encode the message PDU easily.

The first function takes as parameter all the information required to create the requested information element. An information element is formed by the identification field, the criticality field and the value field which is of the ANY type. This value field contains the actual data. Since the function receives as parameter the ASN.1 type that should be encoded into this value field, it can invoke the correct asn1c generated function, Encode_Type_to_ANY(), to do so. This function takes an ASN.1 structure type and data, encodes it and stores the result into the given ANY type field.

Algorithm 1: Encoding a message

```
# Generates a new IE structure with the value field containing
# the given type in APER encoded format.
function Generate_New_IE(ID, Criticality, type, data)
    IE = new IE structure
    Set IE ID
    Set IE Criticality
    if type is a container
        for item = each element in data
            IE2 = new IE structure
            Set IE2 ID according to container type
            Set IE2 Criticality according to container type
            # Provided by the asn1c compiler:
            Encode_Type_to_ANY(item.type, item.data, IE2.value)
            Add IE2 into SEQUENCE structure ss
        end for
        # ss now contains all IEs of the container.
        # Provided by the asn1c compiler:
        Encode_Type_to_ANY(ss.type, ss.data, IE.value)
    else
        # Provided by the asn1c compiler:
        Encode_Type_to_ANY(type, data, IE.value)
    end if
    Return IE
end function

# Encodes data from MessageIEs structure
# into Message in APER format.
function Encode_Message(MessageIEs, Message)
    for element = each IE in MessageIEs
        if element (is mandatory) OR (is optional AND present)
            IE = Generate_New_IE(
                element.ID,
                element.Criticality,
                element.type,
                element.data )
            Add IE into SEQUENCE structure in Message
        end if
    end for
end function
```

In cases where the requested information element is a container type, an additional for-loop is required. This is because the container type information element has a value field of type ANY, but that value field will contain a sequence of additional

information element structures, and each of those structures will also contain a value field of type ANY. Because the asn1c compiler generated code does not know what ASN.1 type should be encoded in each ANY type field, there must be either a script generated or manually added function call for each ANY field inside an information element. Therefore, the for-loop will encode each ANY type fields inside the container first, and only after that can the container contents be encoded into the value field of the container type information element.

The method used in the first function will work for container type information elements that are located at the message structure root, that is, they are listed in the information object set that defines the message contents. If the message contains an information element which includes a container structure as an internal element, the function will not work properly. Luckily most container structures are located at the message root level and are properly handled by the algorithm above. There are only a handful of information elements that include a container as an internal element. One example of this is the Reset message. One of the information elements in this message is the ResetType, which is a CHOICE type construct. One of the members of this CHOICE type is a container structure. To handle this and a few other similar cases, exception handling needs to be added to the above algorithms. This is easiest to do as a last step of the implementation.

Algorithm 2: Decoding a message

```
# Decodes APER encoded data from Message value field
# into MessageIEs structure.
function Decode_Message(Message, MessageIEs)
    Decode_ANY_to_Type(Message.value, message_type, message_data)
    # message_data now contains
    # a sequence of all IEs in the message.
    for ie = each IE in message_data
        if ie is optional
            set presence mask
        end if
        Decode_ANY_to_Type(ie.value, type, MessageIEs.data)
        if type is a container
            for ie2 = each IE in MessageIEs.data
                Decode_ANY_to_Type(ie2.value, subtype, MessageIEs.data)
            end for
        end if
    end for
end function
```

Algorithm 2 describes the message decoding process. This is done in the same way as the message encoding in Algorithm 1, but in reversed order. First, the message PDU is decoded to get the sequence of information elements as ANY type fields. Next, each information element is decoded separately according to the information

element identification number. Each decode function only needs to support those information elements that are allowed to be in the respective message. After an information elements is decoded, the algorithm checks if it was a container type information element. If it was, then the information elements from the container are also decoded.

As was the case for the encoding functions, the decoding functions also need to be modified to handle the special cases where an information element contains an internal container structure. These need to be added for exactly the same information elements as in the encoding functions.

It has been demonstrated that the required algorithms are simple enough and the needed information to generate them is easy to extract from the information object set definitions in the S1AP ASN.1 syntax. Therefore, it is appropriate to develop a Perl script that will parse the relevant ASN.1 syntax files and generate the necessary code. Developing a Perl script for this task has two major advantages. First, the amount of messages in S1AP is so high that writing all this code manually would take more time and would be more prone to errors. And secondly, if a new version is developed in the future with a newer version of the S1AP specifications, the Perl script should work as is or with minor modifications.

The case of information elements with internal container structures can be handled by adding exception handling into the script, or by manually adjusting the script generated code. Updating the script for only a few exceptions would add unnecessary complexity and would be a potential source of new errors. Therefore a manual approach has been implemented for now. The negative side of this choice is that the same modifications must be added manually again if a new version is implemented. This would be a potential place for future improvements.

This chapter explained the ASN.1 syntax used in the S1AP specifications and discussed the limitations of the ASN.1 compiler asn1c regarding the use of information object classes. Due to these limitations, the ASN.1 syntax for S1AP needed to be modified without altering the actual structure of the protocol. The methods used to accomplish this were explained. In addition, a Perl script was developed to parse and process the information available from information objects. This script generates code to supplement the code generated by the asn1c compiler and together they provide a complete and usable implementation of S1AP. This implementation is very low level and close to the internal S1AP structures. Using it would require deep understanding of both ASN.1 and the S1AP ASN.1 syntax. For this reason, an API was also developed to provide a simpler, easy to use interface to the S1AP implementation. The next chapter introduces this API.

5 ARF Integration

The open source ASN.1 compiler asn1c produces C source code. This code is strongly tied to the ASN.1 syntax and requires understanding of both ASN.1 syntax and asn1c itself to be properly used. For this reason, a C++ module was developed to act between the ARF software and asn1c generated components. It hides all details about ASN.1 syntax and asn1c specific functions. This C++ module provides a much more user friendly interface to the S1AP functionality. It provides simple, well documented C++ function calls to create, decode and encode S1AP messages. The module is named as the S1 Application Protocol API.

This chapter presents the architecture and implementation of this API. In addition, the integration of this API into the ARF software is demonstrated. This includes the development of a new module into the ARF software to simulate an MME. The main purpose of this module is to act as a S1AP communication partner with the existing eNodeB implementation within ARF. This enables the development of actual S1 interface functionality into ARF. Lastly, the testing process of the implemented software is discussed.

5.1 The S1 Application Protocol API

As described earlier in chapter 4, S1AP consists of a set of elementary procedures. Each of these elementary procedures contain an initiating message, and optionally a successful outcome and unsuccessful outcome message. Each S1AP message contains a specified set of mandatory and optional information elements. Most of these information elements are also used in other messages. An information element may also act as a container for more information elements.

Files and folders on a computer drive partition would be an apt analogy. The partition root acts like the PDU of the S1AP message and it can contain files and folders, which represent the information elements. A file contains some data, and a folder can contain more files or folders. A folder thus acts as an information element container. The ASN.1 syntax then tells what files and folders are allowed to exist and where, and what is the allowed content structure of each file.

In the S1 Application Protocol API, each S1AP elementary procedure is represented by a C++ class named after the elementary procedure. Each instance of such a class represents a single S1AP message belonging to the relevant elementary procedure. Each of these elementary procedure specific classes inherit the same interface class, IElementaryProcedure, that provides the functions common to all elementary procedures. These include functions to query the class (class 1 or class 2), procedure code, criticality and message type of an elementary procedure object. There is a function call to perform encoding and provide the S1AP message as an APER encoded byte buffer. There is also a function call to return the pointer to the underlying information element structure. This enables direct access to the information element data of the message object, should that be required.

Each elementary procedure specific class then provides services specific to that elementary procedure. In practice, the only difference between elementary procedures

is the types of information elements they contain, and whether the successful outcome or unsuccessful outcome message types are supported. So each elementary procedure class provides the necessary means to create a new instance of that class with the desired information element content. This is implemented by providing constructors which take suitable data as arguments, and additional functions to further set up the contents of the message's information elements. This provides the means to create new S1AP messages from scratch.

Another method to create elementary procedure objects is provided by the elementary procedure factory class. This class provides a function that will take in an APER encoded byte array, decode it according to ASN.1 APER decoding rules [12], and provide the corresponding elementary procedure object. Each elementary procedure class provides appropriate functions to easily read data from the information elements contained in a message.

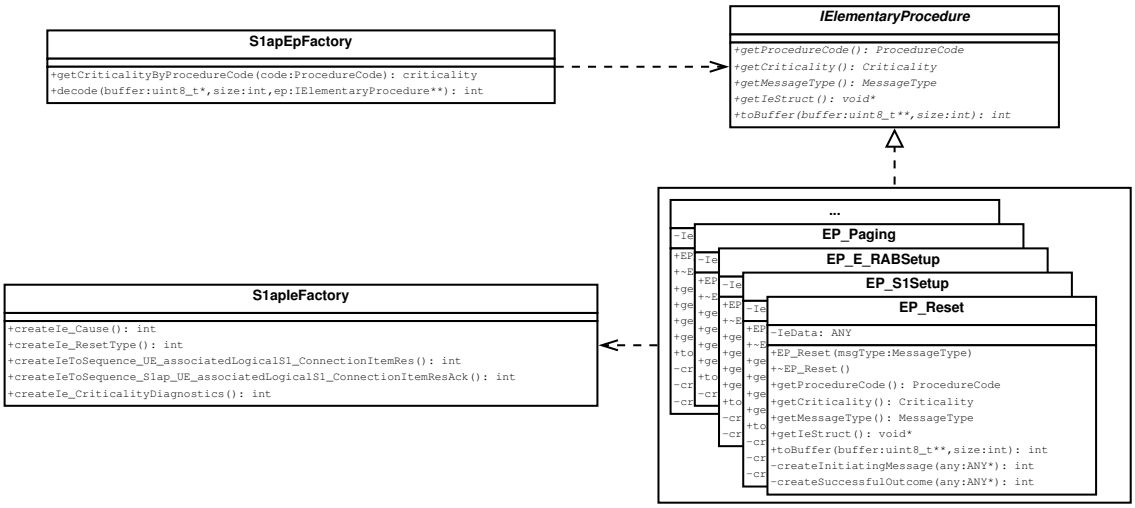

Figure 7: Class diagram for the S1 Application Protocol API.

Figure 7 depicts the class structure of the S1 Application Protocol API. The structure has three main elements, the S1apEpFactory, the S1apIeFactory and the elementary procedure classes. The elementary procedure classes all inherit the interface class IElementaryProcedure, and each elementary procedure in the S1AP specifications has their own class. The S1apEpFactory is an elementary procedure factory. It is able to construct a new instance of any elementary procedure class from an APER encoded byte array, like mentioned above. This is meant to be used when a node receives a new S1AP message and needs to decode it to understand the contents. The S1apIeFactory is an information element factory. It provides support functions to create information element data structures that conform to the relevant ASN.1 syntax.

The information element factory class is used by all the elementary procedure classes to construct their information elements. Since many information elements are used by multiple elementary procedures, it makes sense to provide a centralized collection of functions to generate them. Another option would have been to integrate all information element processing to each elementary procedure class, but this would

have resulted in the same code being written in several places. This is clearly inefficient and increases the chance of errors. By providing separate functions to manage information elements, all implementation details and possible constraint information can be handled in a single place, the relevant function implementation. This way, the caller of the function only needs to be concerned with the value of the data to be stored in the information element, and not the implementation or storage format.

5.2 The S1 Application Protocol logic

To make use of the S1AP implementation in the ARF software, both sides of the S1-MME interface are needed first. On the E-UTRAN side, the ARF already has an eNodeB implementation, but on the EPC side, the ARF did not yet have any MME functionality. As described in the introduction, the goal of this work is to act as an enabler in the further development of the ARF software. The S1AP implementation is only one part of a complicated whole.

To deal with the lack of any working MME node implementation, a new MME module was implemented. The aim in the development of the MME node was to provide a simple node implementation to act as an S1AP communication partner mainly for proof-of-concept and testing purposes. The module was kept otherwise quite simple, but it contains a sophisticated connectivity interface. This interface is implemented with SCTP sockets and the Linux epoll facility, and it allows the MME node to connect to potentially hundreds of different eNodeB nodes. The socket interface is also fully non-blocking allowing the MME node to simultaneously monitor for new incoming connections, receive S1AP messages and being capable of sending S1AP messages as needed. This flexibility in the connectivity interface of the MME node is not really needed in this work, but it will prove to be very useful if the node implementation is developed further in the future.

To the existing eNodeB implementation, a new connectivity interface towards the MME was also needed. For this purpose, a new class called MMEClient was implemented. This class contains a non-blocking SCTP socket implementation that is capable of connecting to a single MME. When connected, the socket can be simultaneously used to constantly listen to incoming S1AP messages, and send S1AP messages as needed.

According to LTE network architecture, an eNodeB can connect to multiple MME nodes. This functionality was not implemented in this work, but it was kept in mind when designing the MMEClient class. The eNodeB implementation can be easily modified to create multiple instances of the MMEClient class, and each can be used to connect to a different MME node. If this is done, some kind of logic also needs to be added to manage the multiple connections.

By keeping the S1AP related functionality in their own classes, the modular structure of the ARF implementation was preserved. Figure 8 depicts a simple class diagram of the S1AP communication process between the eNodeB and MME. On both sides of the interface, both the MME class and the MMEClient class are using the S1AP implementation. Both are using the S1apEpFactory class to decode

incoming S1AP messages and both are using IElementaryProcedure interface class to generate new S1AP messages to send.

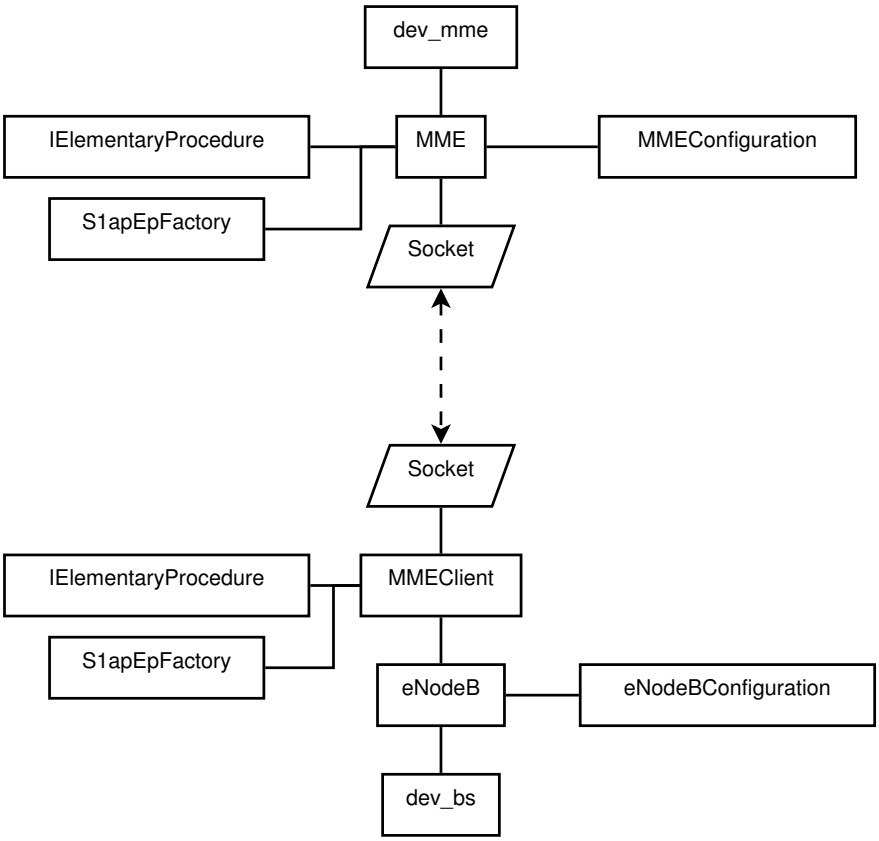

Figure 8: Simple class diagram of the S1AP communication process between the ARF base station process and the MME.

Implementing the functional logic of the S1-MME interface was not the main objective of this work. However, some of the basic functionality was implemented. In the ARF software, when an eNodeB node is started, it will now initiate contact with an MME node during start-up. The eNodeB process will read the network address of the MME from its configuration file. This configuration file can also disable all MME related functionality if needed. The eNodeB contacts the MME by creating a S1SetupRequest initiating message and sending it to the MME. The MME will receive the message, decode it, check the contents and respond by creating a S1SetupResponse successful outcome message and sending it to the eNodeB. These messages create an S1 association between the nodes and state information related to it is stored in both nodes. Currently, if any other messages are received, or a received message contains errors, both nodes will respond with a generic ErrorIndication initiating message. This is a class 2 message and thus has no response message. This simple implementation of the S1 Application Logic provides a good foundation for future development work.

5.3 Testing

Testing is a critically important part of any software development project and should be planned before starting any implementation work. Test-driven development is a particularly useful and recommended process in software development. It relies on very short development cycles where requirements are turned into specific test cases. After specifying and implementing the tests, the actual implementation work is done to the point that all tests are passed and a new cycle can then begin. In this work however, a major part of the code base is generated by the asn1c compiler and Perl scripts. This isn't very suitable to the short development cycle principle of a test-driven development process. For this reason, the S1AP implementation was treated as a single development cycle and the S1 Application Protocol API implementation as a second cycle. Test cases were designed and implemented for both phases.

The ARF software uses Google Test framework to implement testing for its various different components. For consistency, The same test framework was used to implement the new test cases for all S1AP related functionality. The test cases for the S1AP implementation mainly focus on the encoding and decoding functionality generated by the asn1c compiler. They encode and decode different information elements and compare and check the results. The test cases for the S1 Application Protocol API can be divided into two categories: message testing and information element testing. The message tests focus on creating different types of messages, then encoding and decoding them and comparing the message contents to expected values at different stages. The information element tests focus on creating new information element structures and checking any relevant value constraints on different data elements.

There is the possibility that the asn1c compiler would produce a faulty encoder implementation in such a way, that the decoder would include the same fault and cancel it out. In such a situation, the encoder would produce non-compliant output but it would be hard to detect without a 3rd party decoder. To guard against such a scenario, outside sources were also used to obtain example encodings which were used as reference values [25]. Test cases were implemented that recreate these example data structures, encode them, and compare the results to the reference values.

6 Summary

Learning ASN.1 has been a significant part of this implementation work. A good understanding of ASN.1 was required to understand the structure of S1AP messages and without this understanding, this work would have been impossible to carry out. However, ASN.1 is also used in many other specifications. Thus, the knowledge of ASN.1 can be used directly in other projects involving ASN.1.

The open source ASN.1 compiler, asn1c, was used in this work because it was the only viable free alternative. Since it was also open source, the availability of the source code was a significant benefit in case there were any issues encountered. Any potentially required bug fixes, workarounds or alterations could be implemented right away instead of being dependent on a 3rd party to provide such support. However, the asn1c compiler was not without its negative attributes. The compiler itself has not been actively maintained for a few years now. This did not presented any notable problems in the practical implementation work, for the compiler has been left in good working order. The main problem with asn1c was the fact that it did not provide any support for information object classes. This led to the necessity to alter the S1AP ASN.1 syntax into a format compatible with the asn1c compiler. In addition, custom Perl scripts were required to parse and process the information contained within the information object classes. Since the S1AP specifications used information object classes in a relatively simple and consistent way, the development of these Perl scripts proved to be a manageable task considering the scope of this work.

Similarly to ASN.1, the knowledge and experience gained from learning to use the asn1c compiler can be used directly in any other implementation projects involving ASN.1 specifications. This is especially true for many other 3GPP specifications which not only use ASN.1 to specify protocol message structures, but also re-use the exact same information object class structures as well. One good example of this is the X2 Application Protocol (X2AP) used on the X2 interface between eNodeBs [26]. In these cases, the Perl scripts developed for this work could be re-used with only minor modifications.

The goal of this work was to develop a working S1AP implementation and integrate it to the existing ARF software. This has been accomplished. Since the code generated on the basis of the S1AP ASN.1 syntax alone is quite difficult and complex to use, an additional API was also developed to mask this complexity and asn1c specific behaviour. This API successfully provides an easy to use, well documented interface for the S1AP implementation.

The ARF integration was accomplished by developing new classes into the ARF that implement the required framework for the new S1-MME interface functionality. These new classes also implement basic S1 session set up during the ARF start up procedure.

All new code was tested. Initial testing found some problems in the S1 Application Protocol API implementation regarding the way some information element containers were being encoded. These issues were fixed and all test cases now pass.

To continue the ARF software development, and to properly make use of the S1AP implementation, the next logical step would be to continue developing the functional

logic of the S1-MME interface. The current establishment of the S1 session during the ARF start up procedure is only the first step. More non-UE associated signalling should be implemented next. Also the framework for UE associated signalling should be planned. This framework would need to be able to manage and store all necessary state information related to UEs that connect to the eNodeB. Another potential development target would be to implement the X2 interface. This interface allows direct communication between two eNodeBs.

References

[1] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Architecture description. TS 36.401 version 10.3.0 Release 10, Sept. 2011.

[2] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 general aspects and principles. TS 36.410 version 10.2.0 Release 10, Sept. 2011.

[3] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 layer 1. TS 36.411 version 10.1.0 Release 10, June 2011.

[4] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 signalling transport. TS 36.412 version 10.1.0 Release 10, June 2011.

[5] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 Application Protocol (S1AP). TS 36.413 version 10.5.0 Release 10, March 2012.

[6] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 data transport. TS 36.414 version 10.1.0 Release 10, June 2011.

[7] ITU-T. Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU-T Recommendation X.680, July 2002.

[8] ITU-T. Information technology – Abstract Syntax Notation One (ASN.1): Information object specification. ITU-T Recommendation X.681, July 2002.

[9] ITU-T. Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification. ITU-T Recommendation X.682, July 2002.

[10] ITU-T. Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications. ITU-T Recommendation X.683, July 2002.

[11] ITU-T. Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). ITU-T Recommendation X.690, July 2002.

[12] ITU-T. Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER). ITU-T Recommendation X.691, July 2002.

[13] ITU-T. Information technology – ASN.1 encoding rules: XML Encoding Rules (XER). ITU-T Recommendation X.693, Dec. 2001.

[14] ITU-T. Information technology – ASN.1 encoding rules: Specification of Octet Encoding Rules (OER). Recommendation ITU-T X.696, Aug. 2015.

[15] Dubuisson, O. ASN.1 — Communication Between Heterogeneous Systems. Morgan Kaufmann, October 2000.

[16] Larmouth, J. ASN.1 Complete. Morgan Kaufmann, October 1999.

[17] Kerttula, J., Malm, N., Ruttik, K., Jäntti, R. and Tirkkonen, O. Implementing TD-LTE as Software Defined Radio in General Purpose Processor. In The Software Radio Implementation Forum, SRIF'14, pages 61–67, Chicago, IL, USA, August 2014. ACM.

[18] Malm, N. Ultra-reliable Network-controlled D2D. Master's Thesis, Aalto University School of Electrical Engineering, Department of Communications and Networking, Espoo, 2015.

[19] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 Application Protocol (S1AP). TS 36.413 version 8.7.0 Release 8, Sept. 2009.

[20] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 Application Protocol (S1AP). TS 36.413 version 12.6.0 Release 12, June 2015.

[21] OpenAirInterface. OpenAirInterface website. http://www.openairinterface.org/, 2016.

[22] Open Source ASN.1 Compiler by Lev Walkin. The asn1c website. http://lionet.info/asn1c/compiler.html, 2016.

[23] Tools for ASN.1. The ITU website. http://www.itu.int/en/ITU-T/asn1/Pages/Tools.aspx, 2016.

[24] IETF. Stream Control Transmission Protocol. RFC 4960, Sept. 2007.

[25] Kreher, R. and Gaenger, K. LTE Signaling: Troubleshooting and Optimization. Wiley, December 2010.

[26] 3GPP. LTE; Evolved Universal Terrestrial Radio Access Network (E-UTRAN); X2 Application Protocol (X2AP). TS 36.423 version 10.7.0 Release 10, Sept. 2013.