



FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

**Otto Poikajärvi**

**Design and development of protocol log analyzer for cellular modem**

Master's Thesis  
Degree Programme in Computer Science and Engineering  
November 2021

**Poikajärvi O. (2021) Design and development of protocol log analyzer for cellular modem.** University of Oulu, Degree Programme in Computer Science and Engineering. Master's Thesis, 65 p.

## **ABSTRACT**

Telecommunications protocols and cellular modems are used in devices to facilitate wireless communication. Cellular modems produce log files, which have to be analyzed by engineers when issues occur. Performing the analysis for large logs manually can be very time consuming, thus different approaches for trying to automate or simplify the process exist.

This thesis presents design and development for a cellular modem log analysis tool. The tool is designed to take into account peculiarities of telecommunications protocols and cellular modems, especially of 5G New Radio Radio Resource Control protocol. A notation for defining analysis rules used by the tool is presented to be used alongside the tool.

The developed tool is a proof-of-concept, with focus being on how the tool performs the analysis and how the notation can be used to define the wanted analysis rules. The features of the notation include defining expected content of protocol messages and order of log message sequences. The tool performs well with artificial modem logs, though some flaws in the notation are recognized. In the future, the tool and the notation should be updated with support for real cellular modem logs and evaluated in field use cases by cellular modem engineers.

**Keywords:** telecommunications, wireless communication, log analysis, cellular modem, 5G New Radio, metasyntax

**Poikajärvi O. (2021) Matkapuhelinmodeemien lokitiedostojen analysointityökalun suunnittelu ja toteutus.** Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 65 s.

## **TIIVISTELMÄ**

Tietoliikenneprotokollia ja matkapuhelinmodeemeja käytetään laitteissa langattoman tiedonsiirron mahdollistamiseksi. Matkapuhelinmodeemit tuottavat lokitiedostoja, joita insinöörien täytyy analysoida ongelmatilanteissa. Suurten lokitiedostojen analysointi manuaalisesti on työlästä, joten on olemassa keinoja prosessin automatisointiin tai yksinkertaistamiseen.

Tämä työ esittelee suunnitelman ja toteutuksen matkapuhelinmodeemin lokitiedostojen analysointityökalulle. Työkalun suunnittelussa on otettu huomioon tietoliikenneprotokollien, erityisesti 5G New Radion radioresurssien hallintaprotokollan (RRC), ja matkapuhelinmodeemien erikoisuudet. Merkintäsäännöstö, jolla voidaan määritellä analyysisäännöt, esitellään työkalulle.

Kehitetty työkalu on karkea prototyyppi. Kehityksessä keskitytään työkalun analyysiominaisuuksiin ja mahdollisuuksiin käyttää merkintäsäännöstöä määrittämään halutut analyysisäännöt. Merkintäsäännösten ominaisuuksiin kuuluu odotettujen lokiviestien sisällön ja järjestyksen määrittely. Työkalu suoriutuu keinotekoisien modeemilokitiedostojen kanssa hyvin, mutta joitain vikoja merkintäsäännöstöstä havaittiin. Tulevaisuuden kehitystä ajatellen työkalu kannattaisi päivittää toimimaan aitojen matkapuhelinmodeemien lokitiedostojen kanssa, että sen kykyä suoriutua aidoista käyttötilanteista voitaisiin arvioida.

**Avainsanat:** tietoliikenne, langaton tiedonsiirto, lokianalyysi, matkapuhelinmodeemi, 5G New Radio, metakieli

# TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

FOREWORD

ABBREVIATIONS

|        |                                                         |    |
|--------|---------------------------------------------------------|----|
| 1.     | INTRODUCTION.....                                       | 7  |
| 2.     | BACKGROUND.....                                         | 9  |
| 2.1.   | Telecommunications protocols.....                       | 9  |
| 2.2.   | 3GPP 5G cellular network.....                           | 11 |
| 2.2.1. | Radio Resource Control Protocol.....                    | 15 |
| 2.3.   | Log parsing.....                                        | 18 |
| 2.4.   | Failure detection.....                                  | 21 |
| 2.4.1. | Heuristic log file analysis by Mariani and Pastore..... | 22 |
| 2.5.   | Metasyntax and data formatting.....                     | 24 |
| 2.5.1. | ASN.1.....                                              | 26 |
| 2.6.   | Summary.....                                            | 28 |
| 3.     | REQUIREMENTS.....                                       | 29 |
| 3.1.   | Utility requirements.....                               | 29 |
| 3.2.   | Analysis requirements.....                              | 30 |
| 3.3.   | Technical requirements.....                             | 30 |
| 4.     | DESIGN.....                                             | 32 |
| 4.1.   | Architecture.....                                       | 32 |
| 4.2.   | Log parser.....                                         | 33 |
| 4.3.   | Rule notation.....                                      | 34 |
| 4.3.1. | Event structure in the notation.....                    | 37 |
| 4.3.2. | Specifying reactions.....                               | 37 |
| 4.3.3. | Chaining different operators and keywords.....          | 37 |
| 4.4.   | Rule reader.....                                        | 38 |
| 4.5.   | Analyzer.....                                           | 41 |
| 4.6.   | UI.....                                                 | 42 |
| 4.7.   | Summary.....                                            | 44 |
| 5.     | IMPLEMENTATION.....                                     | 45 |
| 5.1.   | Architecture.....                                       | 45 |
| 5.2.   | Log parser.....                                         | 46 |
| 5.3.   | Rule reader.....                                        | 47 |
| 5.4.   | Analyzer.....                                           | 48 |
| 5.5.   | Main and UI.....                                        | 50 |
| 5.6.   | Summary.....                                            | 52 |
| 6.     | RESULTS.....                                            | 53 |
| 6.1.   | Evaluation preparation.....                             | 53 |
| 6.2.   | Creating the rule set.....                              | 53 |
| 6.3.   | Analyzing the artificial log.....                       | 57 |
| 6.4.   | Issues and possible solutions.....                      | 59 |
| 7.     | DISCUSSION.....                                         | 61 |
| 8.     | CONCLUSION.....                                         | 63 |
| 9.     | REFERENCES.....                                         | 64 |

## **FOREWORD**

This thesis was written while working at MediaTek Wireless Finland Oulu office. I would like to thank MediaTek for the opportunity to write this thesis, the entire Protocol Team at MediaTek Oulu and especially Pasi Laitinen for technical assistance and support, and my thesis supervisor Ella Peltonen.

Oulu, 9.11.2021

Otto Poikajärvi

## **ABBREVIATIONS**

|      |                                    |
|------|------------------------------------|
| BNF  | Backus-Naur Form                   |
| EBNF | Extended BNF                       |
| NR   | New Radio                          |
| PDU  | Protocol Data Unit                 |
| RRC  | Radio Resource Control             |
| UE   | User Equipment                     |
| UI   | User Interface                     |
| 3GPP | 3rd Generation Partnership Project |

# 1. INTRODUCTION

Wireless communication is used constantly in our daily lives. Smartphones depend on their ability to connect to internet services to provide us entertainment, ways to communicate, and even banking services. Wireless communication requires cellular modems, which transform digital information within devices to radio waves sent through the air. These cellular modems are built according to different standardized telecommunication protocols, for example protocols specified by 3GPP used in 4G and 5G communication. As cellular modems are complicated devices, creation of a perfect cellular modem is impossible. Sometimes cellular modems fail, and engineers have to find out what went wrong. This is when cellular modem logs come into play. The cellular modem writes different type of information about events preceding the failure in the cellular modem log. It can be necessary for the engineers to analyze these logs manually, sometimes going through hundreds of stored log messages. This process can be very time consuming and can require effort from specific expert engineers with in-depth knowledge about the inner workings of the cellular modem. The engineers' time would be better spent on developing fixes and new features for the cellular modem, instead of going through countless cellular modem logs. Cutting down time spent on log analysis can both improve the efficiency of engineer work and mean that engineers can spend more time with more rewarding and interesting tasks.

The goal of this thesis is to chart what kind of methods could be used to ease the workload of log analysis and develop an analysis tool for 5G cellular modem logs. The logs are message logs, where each log message corresponds to different action of the cellular modem. The tool is named Protocol Log Analyzer. Protocol Log Analyzer must be able to work with different log formats and support different message rules. The engineers know these message rules, but Protocol Log Analyzer must include a method for them to insert that knowledge into it. It will locate where in the log rule breaks or other interesting events can be found, based on the previous knowledge of the engineers. Most importantly, Protocol Log Analyzer must help the engineers with their log analysis activities, either by allowing for other people than the expert engineers to analyze logs, or by making it easier and faster for the experts to locate issues from logs. In practice, learning how to use Protocol Log Analyzer and integrating it to engineers' daily work should be easier than continuing to analyze logs manually.

This thesis presents requirements for Protocol Log Analyzer and a prototype that takes specialties of telecommunication protocols into account. Alongside Protocol Log Analyzer, a first version of a notation, meant for engineers to write rules about log analysis, is presented. The notation is based on previously existing metasyntax notations, with Extended Backus-Naur Form [1] serving as a major inspiration, though the developed notation is more specialized for defining information expected to be present in cellular modem logs.

Chapter 2 of this thesis provides background information on already existing log analysis tools and researches what kind of data cellular modems could write in their logs, including presenting modern telecommunications protocols. Chapter 3 identifies the requirements for Protocol Log Analyzer based on wishes of engineers working on cellular modems. Chapter 4 presents the design for Protocol Log Analyzer, how it is split into different modules, and defines the developed rule notation. Chapter 5 presents the implementation, including the class structure. Protocol Log Analyzer is

evaluated and results discussed in Chapter 6. Chapter 7 includes discussion on what was achieved and how development could proceed. Chapter 8 sums up the thesis.



## 2. BACKGROUND

### 2.1. Telecommunications protocols

Understanding what kind of messages are exchanged in telecommunication is essential for understanding how cellular modem logs can be analyzed. These messages are defined in different kinds of telecommunication protocols. Sharp [2, pp. 1-5] defines characteristics of protocols as a set of formal rules for information exchange and that following those rules is mandatory for successful information exchange. When it comes to communication between computer systems, all parties using the same protocol is requisite for successful communication. Protocol defines all possible messages and legal sequences of messages. It can be analyzed as a kind of language with its own set of sentences and symbols [2]. Hercog [3, p. 16] splits protocol language syntax into three sections, abstract syntax defines the available messages, transfer syntax defines the format of protocol messages, and supermessage syntax defines what kind of sequences of messages are allowed and when messages may or must be sent by devices using the protocol. Hercog [3, p. 17] notes that the sequence of protocol messages is the most important part of following a protocol. This means that it is something that should be paid extra attention to when trying to find faults on why message exchange fails using a telecommunication protocol.

As Sharp [2, p. 64] explains, communication systems often use layered architectures. An example of a layered protocol architecture with different components highlighted can be seen in Figure 1. In these protocol stacks, lower layer provides services for the upper layer. Holzmann [4, pp. 27-32] describes how layered architecture can be used to split complicated communication tasks to smaller subtasks and that in a layered architecture lower layers provide ‘virtual’ communication channels that allow specific layers to communicate with their corresponding layers in other devices. When a user of communication system sends information from N-th layer, it first travels ‘down’ through each lower layer of the sender, through physical circuit between the sender and the receiver, and then ‘up’ the layers on the receiver’s end until the N-th layer is reached. Holzmann [4, p. 31] calls the entities existing in the same layer peer entities and connection between these entities a peer protocol. Boundary between adjacent layers is called an interface. Holzmann [4, p. 31] also notes that in communication systems only the peer protocols must be standardized between the communicating systems, and implementation of inter-layer interfaces can differ between the sender and the receiver. The messages exchanged by peer entities are called Protocol Data Units (PDUs) [2, p. 72]. Messages between adjacent layers in the interface are called primitives [2, p. 94]. These primitives facilitate information exchange between upper and lower layers. PDU of an upper layer is called Service Data Unit (SDU) in the primitive and lower layers [2, p. 72]. Lower layers do not process or analyze information within SDUs [3, p. 72]. When analyzing exchange of protocol messages, the focus is usually on a specific layer. As message contents of upper layers are not relevant for the functionality of lower layer, they can be ignored when analyzing message exchange on the lower layer.

Common example of a layered architecture is the OSI Reference Model, which is visualized in Figure 2. Sharp [2, pp. 64-65] writes that important features of the OSI Reference Model is not the exact functionality of the seven available layers, but instead the layering principle and the notation introduced. Another example of a protocol stack

is Internet protocol suite, which has five different layers. In it, the three upper most layers of OSI Reference Model are considered as one Application layer [2]. According Hercog [3, pp. 70, 77-78], the uppermost layer being called application layer is usual. The application layer is composed of the users of the communication system using the protocol stack. The lowest layer is then physical and it is responsible for communication through the real physical communication channel.

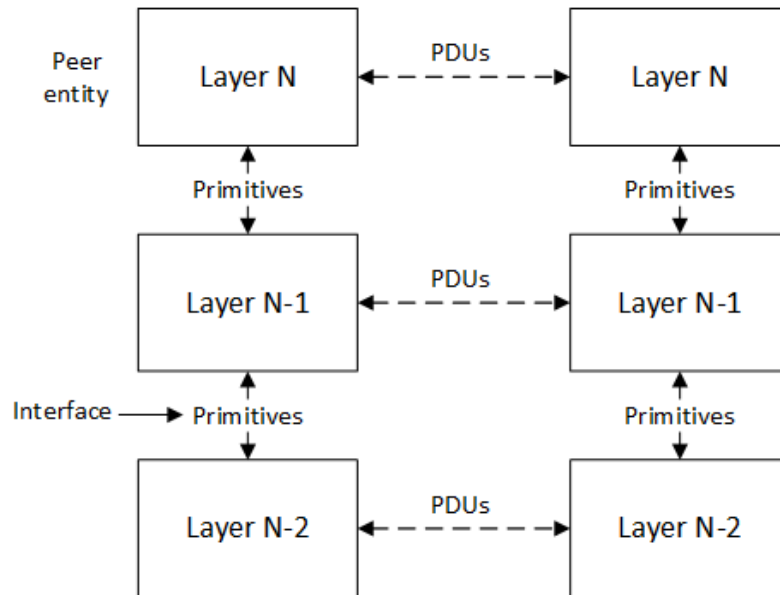


Figure 1: Protocol stack components

| Number  | Name         |
|---------|--------------|
| Layer 7 | Application  |
| Layer 6 | Presentation |
| Layer 5 | Session      |
| Layer 4 | Transport    |
| Layer 3 | Network      |
| Layer 2 | Data Link    |
| Layer 1 | Physical     |

Figure 2: OSI Reference Model, adapted from Hercog [3, p. 77]

Hercog [3, pp. 56-58] presents a model of protocol entity and lists its requirements. A protocol entity can measure time, transmit and receive messages, memorize messages and values, interact with users of the protocol, and follow actions defined by the protocol. In order to fulfill these requirements the process entity requires a processor and memory. In order to measure time the process entity uses a timer. It can

activate or deactivate the timer when the protocol so requires. The timer expires after a predefined time, which can cause the protocol entity to react as defined in the protocol used. The protocol entity might not be able to process events such as new messages or timer expiries immediately. A queue in memory is used to store these inputs until the protocol entity can handle and react to them [3]. Example of timers used in protocol is Sharp's [2, pp. 285-287] presentation of ISO Class 4 Transport Protocol. The protocol uses several timers in order to resist different type of failures including a window timer that is used to detect unindicated connection failures by making sure at least some PDUs are being sent during a specific interval. Timers can be relevant when analyzing telecommunication message logs. For example, if each message has a timestamp attached to it, those timestamps can be analyzed to see if timers defined in the protocol expire before a protocol entity has answered to some specific message.

Telecommunications protocol implementations have to be tested. Dubuisson [5, pp. 480-481] explains that testing for protocol implementations is done by examining the external reactions of the implementation. The implementation is not taken into account, just how it reacts and answers to specific input messages. If the implementation follows the protocol according to the tests, then it can work with other systems that implement the same protocol. TTCN (Tree and Tabular Combined Notation in Dubuisson [5], Testing and Test Control Notation for the newest version 3<sup>1</sup>) is a language designed for describing these protocol tests. TTCN focuses on the PDUs and service primitives. Its test suites are comprised of structures of tables. The test suites indicate possible events and they are either passed or failed depending on if the correct events are observed in the correct order [5]. Engineers use logs as one of their tools in finding out why implementations do not follow protocol when tests fail. The implementation can write to the log its reasons for sending or not sending specific messages, which can guide engineers to right path during issue diagnosis if they engineers can locate the relevant log messages.

## 2.2. 3GPP 5G cellular network

This Chapter elaborates on aspects of modern 5G cellular networks. Understanding the content of a cellular modem log is not possible without understanding which entities communicate with each other and what is relevant for the functionality of a 5G modem. The most relevant telecommunications protocols for cellular modems are defined by 3rd Generation Partnership Project (3GPP). 3GPP was set up on 1998 in order to provide a forum for discussing global standards for WCDMA specification [6]. 3GPP has seven telecommunications standard development organizations as organizational partners (ARIB, ATIS, CCSA, ETSI, TSDSI, TTA, TTC) as seen on their website<sup>2</sup>. 3GPP specifications include radio access, core network and service capabilities, and other telecommunications technologies. 3GPP includes three Technical Specification Groups (TSG): Radio Access Networks (RAN), Services and Systems Aspects, and Core Network and Terminals. Each TSG is then further split into Working Groups, which focus on certain subsection of telecommunications. The TSGs produce protocol specifications as their output.

---

<sup>1</sup> <http://www.ttcn-3.org/>

<sup>2</sup> <https://www.3gpp.org/>

5G is the newest wireless telecommunication technology for mobile phone networks. Holma et al. [7] present the schedule for 5G standardization until 5G deployment was possible. 3GPP's work on 5G started in 2015 but the content of first specification was finished in Release 15 in December of 2017. Deployment of Non-Standalone (NSA) 5G was finally possible with 2018 December version, with Stand-Alone (SA) following in March 2019. NSA and SA architectures are explained with more details below. First commercial 5G networks were then launched in April 2019. According to 3GPP's website<sup>2</sup>, Release 16, which was finalized in June 2020, brought the full 3GPP 5G system to completion. As 5G has now launched properly, future developments with cellular modems focus on it, and so does this thesis.

Toskala and Poikselkä [8] present an overview of 5G architectures. There are two main architecture options, NSA and SA. In NSA, 4G LTE is used as the anchor for the connection existing LTE core (Evolved Packet Core, EPC). In this architecture, 5G radio is only used on the user plane with dual connectivity with LTE, meaning that the UE is simultaneously connected to both LTE and 5G NR. NSA has a few different data routing options, which differ on how user plane is split between LTE and 5G. LTE and 5G nodes in NSA can exchange data between each other or they can both directly connect to the EPC. In SA, the 5G radio is connected to 5G Core Network. SA allows for full 5G functionality and does not need LTE-5G dual connectivity. The rest of the Chapter focuses on SA.

In order to understand failures visible in modem logs, it must first be understood what kind of information is exchanged in those messages and how they affect the functionality of both protocol entities. As defined in TS 38.300 [9], the 5G node connecting to 5G Core Network is either a gNB or an ng-eNB. gNB NG-RAN node provides NR protocol terminations for the UE and ng-eNB provides the same for EUTRA. The 5G nodes are connected to the 5G Core Network via NG interfaces, with control plane connecting to Access and Mobility Management Function (AMF) and user plane to User Plane Function (UPF). The NG-RAN nodes are connected to each other with Xn interface. Xn interface is also split to user plane and control plane, with both having their own protocol stacks. NG-RAN nodes are connected to 5G Core Network with NG interfaces. Figure 3 visualizes different entities and their connections in the 5G SA architecture. The Uu interface between the gNB and the UE is the most interesting interface from the perspective of the cellular modem and this work.

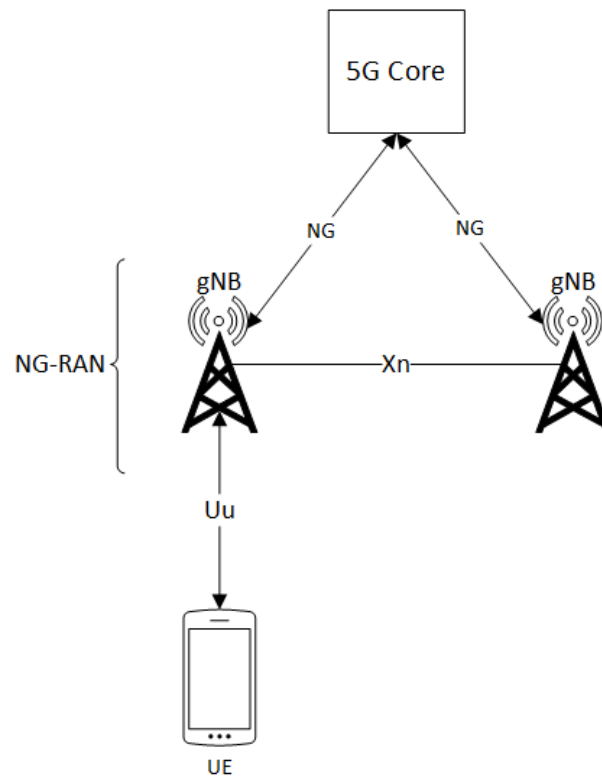


Figure 3: 5G entities and connections, adapted from TS 38.300 [9]

Working Group responsible for radio interface protocols between UE and RAN is RAN2<sup>3</sup>. 3GPP derived radio protocols for 5G NR from 4G LTE but differ due to 5G requirements such as high data rates with low latency [10]. RAN2 specifications related to NR include but are not limited to “TS 38.300 NR; NR and NG-RAN Overall description; Stage-2”, “TS 38.304 NR; User Equipment (UE) procedures in idle mode and in RRC Inactive state” “TS 38.306 NR; User Equipment (UE) radio access capabilities”, “TS 38.321 NR; Medium Access Control (MAC) protocol specification”, “TS 38.322 NR; Radio Link Control (RLC) protocol specification”, “TS 38.323 NR; Packet Data Convergence Protocol (PDCP) specification”, “TS 38.331 NR; Radio Resource Control (RRC); Protocol specification”, and “TS 37.324 Evolved Universal Terrestrial Radio Access (E-UTRA) and NR; Service Data Adaptation Protocol (SDAP) specification”. RRC protocol is the one responsible for configuring the UE with parameters the other protocol layers need and for maintaining connectivity between the UE and the network [10].

Working Group RAN5 focuses on conformance testing of radio interface for the UE, including radio interface protocols defined by RAN2<sup>3</sup>. For 5G NR, RAN5’s specifications include “TS 38.508-1 5GS; User Equipment (UE) conformance specification; Part 1: Common test environment” and “TS 38.523-1 5GS; User Equipment (UE) conformance specification; Part 1: Protocol”, which describe message sequences and message contents that can be used to confirm UE’s conformance to that specific 3GPP release.

As 3GPP’s website<sup>3</sup> explains, 3GPP’s technologies evolve with generations of mobile systems, the newest being 5G, but 3GPP specifications update in Releases instead. Development of different Releases happens in parallel, meaning that the next

<sup>3</sup> <https://www.3gpp.org/>

release is already underway when the previous release is ‘frozen’ and released ready for implementation. The purpose of Releases is to have a stable platform for implementation while simultaneously allowing for addition of new features for future releases. The Releases are backwards and forwards compatible when possible. This is to allow for devices within the network to function without interruptions, e.g. UE built with an older Release can still function when surrounding infrastructure is updated. This also allows the same methods to be used when analyzing modem logs of different releases, as different releases can add more procedures and message fields, but the same basic principles apply.

Henttonen et al. [10] describe the 5G radio protocol stack between a gNB and a UE. The protocols are often split to two different planes, the previously mentioned user plane (UP), and a control plane (CP). When using the OSI Reference model as a reference, the UP of 5G New Radio (NR) has four different Layer 2 sublayers, and a physical layer (Layer 1). The four sublayers are Service Data Adaptation Protocol (SDAP), Packet Data Convergence Protocol (PDCP), Radio Link Control (RLC), and Medium Access Control (MAC). The topmost sublayer SDAP is connected to its counterpart in UPF, though SDAP is not necessary when the connection is NSA. The CP shares PDCP, RLC, MAC and Layer 1 with the UP, but has Radio Resource Control (RRC) on top of PDCP. Non-Access Stratum (NAS) is then on top of RRC. NAS is part of AMF of the 5GC. Protocol stack can be seen in Figure 4.

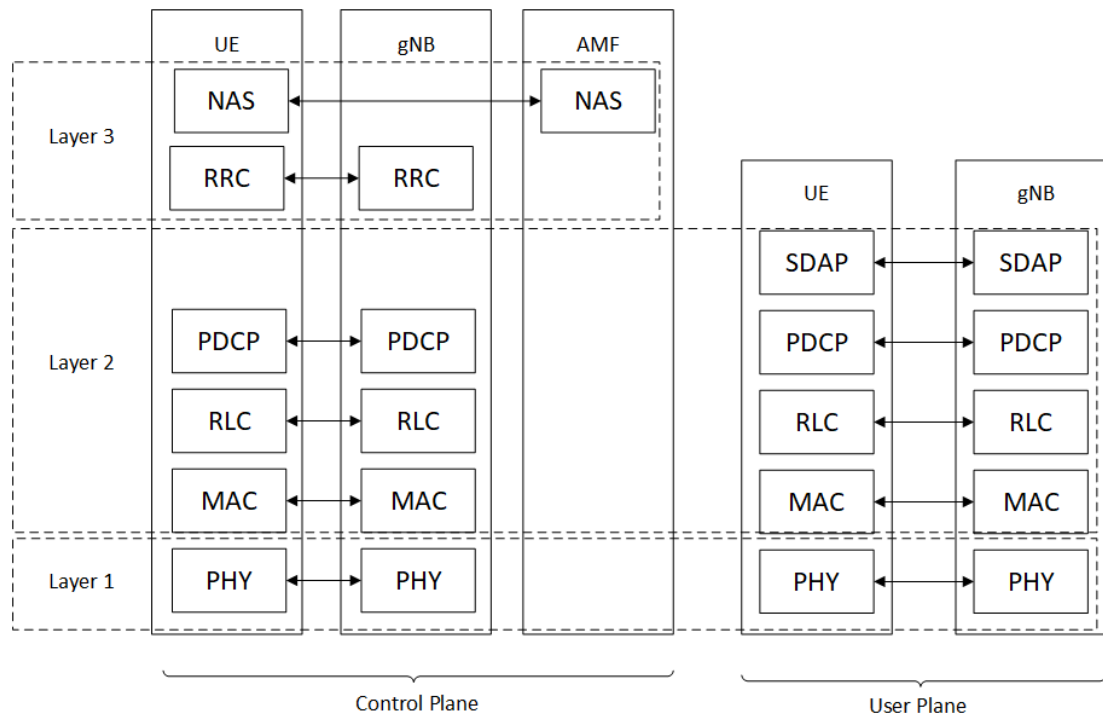


Figure 4: 3GPP 5G Radio Protocol Stack, adapted from Henttonen et al. [10]

There are three layers of channels in 5G New Radio [10]. MAC provides data transfer services with different types of logical channel types [9]. Different type of information is transferred using different logical channels. There are two types of logical channels, control channels and traffic channels. There are four different control channels. Broadcast Control Channel (BCCH) is used for broadcasting system control information, Paging Control Channel (PCCH) for paging messages, Common Control Channel (CCCH) for control information transmission between the UE and the

network, which is used when there is no RRC connection (see 2.2.1 for RRC connections), and Dedicated Control Channel (DCCH) for control information transmission between the UE and the network when there is an RRC connection. Control channels are used for control plane information. There is only one traffic channel, Dedicated Traffic Channel (DTCH). It is used to transfer user information of the user plane. MAC maps these logical channels to transport channels. Transport channels are further mapped to physical channels, which are the lowest channel layer [10].

### 2.2.1. Radio Resource Control Protocol

Radio Resource Control (RRC) protocol is elaborated more in-depth here as an example of a cellular modem protocol. TS 38.331 [11] specifies three possible states for the UE in NR, RRC\_CONNECTED, RRC\_INACTIVE, and RRC\_IDLE. The UE is in RRC\_IDLE state if RRC connection has not been established. In RRC\_INACTIVE and RRC\_CONNECTED the RRC connection has been established, but in RRC\_INACTIVE it is only stored instead of actively used to save power. In RRC\_IDLE, gNB node does not have UE AS context stored, unlike the last serving gNB in RRC\_INACTIVE and the gNB in RRC\_CONNECTED, meaning that in RRC\_IDLE the gNB node is not aware of the UE. In RRC\_IDLE and RRC\_INACTIVE the UE controls its mobility, but in RRC\_CONNECTED mobility is controlled by the network. Possible state transitions between the states are resume, release, release with suspend, and establish, as seen in Figure 5.

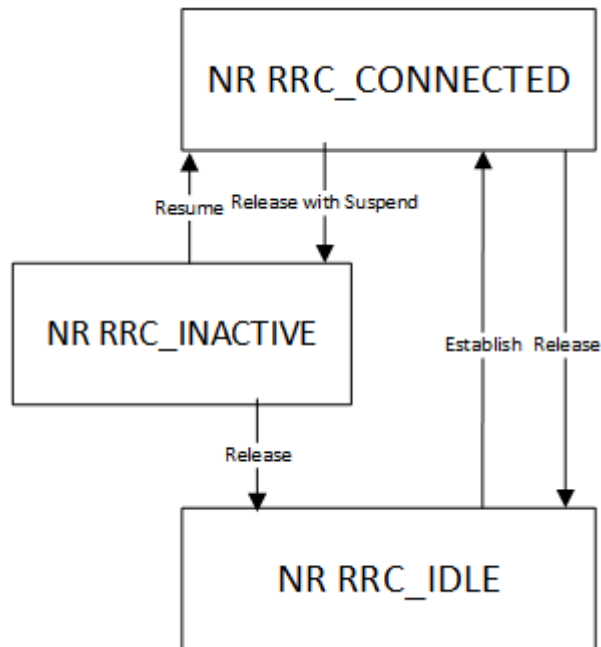


Figure 5: RRC states for the UE in NR, adapted from TS 38.331 [11]

Mobility between radio access technologies (inter-RAT) adds three more possible state transitions, handover, reselection and redirection, visible in Figure 6.

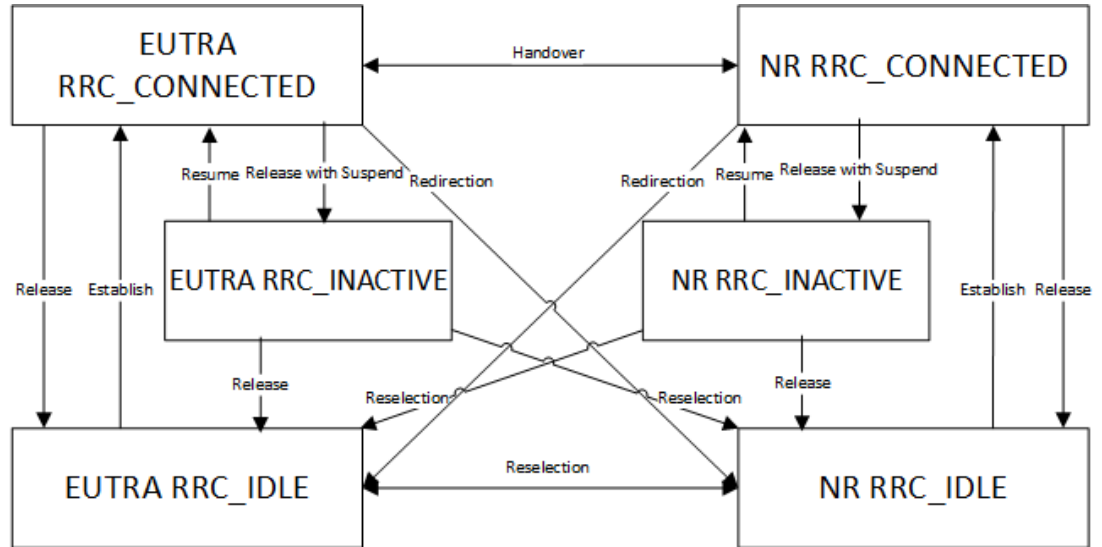


Figure 6: RRC states including mobility to and from NR, adapted from TS 38.331 [11]

#### 2.2.1.1. RRC procedures

Procedures that lead to RRC state transitions have some general requirements. The UE shall process the messages in the receiving order, even if the gNB may initiate the next procedure before the UE's response. If the UE's response message includes a transaction identifier, it should have the same value as the gNB message's transaction identifier had. Purpose of the transaction identifiers is to ensure the UE ignores duplicate RRC messages and that the gNB knows when the UE has finished its procedure to a specific message [10]. Most downlink messages include the transaction identifier, notable exceptions being broadcast messages and messages which cause the UE to move to RRC\_IDLE or RRC\_INACTIVE states. Transaction identifiers can be used when analyzing modem logs to pinpoint which RRC message was never answered to. By identifying specific sets of RRC messages in modem logs, specific procedures can also be identified and compared to successful cases.

Henttonen et al. [10] say connection control can be considered the primary task of RRC because it allows the UE to transmit and receive data. They also note that RRC mobility procedures are done by reusing connection control procedures. Connection control actions presented include connection establishment, capability reporting, connection reconfiguration, and connection release. Certain connection control procedures can only start when the UE is in a specific RRC state.

Examples for connection control procedures are RRC connection establishment and RRC reconfiguration. RRC connection establishment described in TS 38.331 [11] is a connection control procedure in which the UE in RRC\_IDLE sends *RRCSetupRequest* to the gNB, which responds with *RRCSetup*. After the UE has processed *RRCSetup* and sent *RRCSetupComplete*, the procedure will finish with UE entering RRC\_CONNECTED state. The successful procedure is visualized in Figure 7.



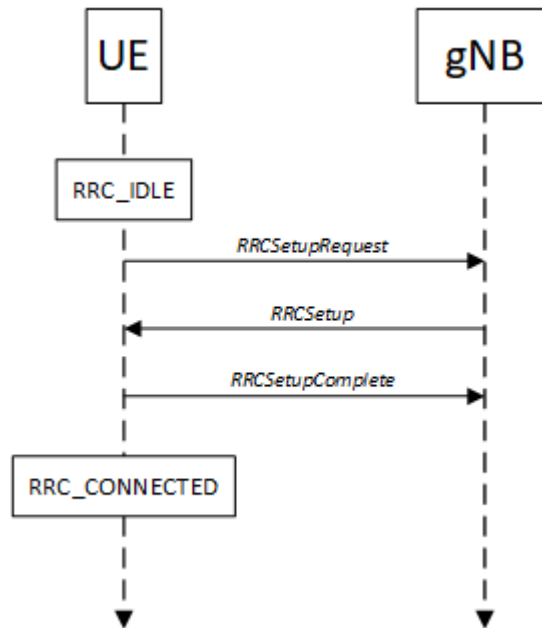


Figure 7: Successful RRC connection establishment, adapted from TS 38.331 [11]

RRC reconfiguration procedure [11] can start when the UE is in RRC\_CONNECTED. The gNB initiates the procedure when it modifies the details of an RRC connection. Procedure starts by the gNB sending *RRCReconfiguration*. The UE reacts to *RRCReconfiguration* accordingly and in the case of a successful reconfiguration responds with *RRCReconfigurationComplete*. Initial AS security activation and first RRC reconfiguration have to happen following RRC connection establishment until user data can be transferred [10].

TS 38.523-1 [12] includes a test procedure for RRC connection establishment followed by initial AS security activation and RRC reconfiguration. The test procedure chains different test purposes together with different RRC reconfiguration following each other. In order to pass the conformance testing, the UE tested must answer to messages sent by the testing environment's system simulator according to previously defined rules. E.g. when activating AS security, the system simulator sends both *SecurityModeCommand* and *RRCReconfiguration* messages. The UE then must answer with *SecurityModeComplete* and *RRCReconfigurationComplete* messages. The same elements tested in the test procedure can be extracted from modem logs. If the log includes messages of RRC connection establishment, initial AS security activation, and RRC reconfiguration procedures, the log can be further analyzed in order to find possible divergences when comparing to the example case.

#### 2.2.1.2. RRC messages

TS 38.331 [11] specifies the contents of each RRC message using Abstract Syntax Notation One (ASN.1). ASN.1 is a notation used for describing messages between different computing applications [5]. ASN.1 and its encoding rules are platform agnostic, which makes it suitable for a wide array of applications, including 3GPP specifications. RRC messages often include several fields, which types are smaller information elements that are also defined in the specification. These RRC information elements can be reused in several different RRC messages. Not all fields in RRC messages are always present and the specification defines which fields are optional,

conditionally mandatory, or mandatory. Specification defines purpose, direction and logical channel for each RRC message.

Certain fields in RRC messages or information elements contained within them are limited in when the network can configure or release them. For example, information element *ServingCellConfig* can only include field *uplinkConfig* when the network has configured field *uplinkConfigCommon* separately in information element *ServingCellConfigCommon* or *ServingCellConfigCommonSIB* for the UE [11]. Configuring *uplinkConfig* without the previous requirement is against the specification and may lead to faulty functionality.

Error handling for faulty messages is defined for RRC messages [11]. Error handling procedure differs for different logical channels. For messages using BCCH, CCCH or PCCH, invalid abstract syntax handling is used when one or more fields in the message are set to values, which are not spare, reserved, or extended values. This causes the error handling to happen at message level. In these cases, the message is ignored. In the case of a spare, extended or reserved value in a field of the message, the UE either replaces it with a default value or handles the message as if the field was not present. If the entire field in the message is defined as spare or reserved, the UE handles the message as if the field was not present. Handling of missing mandatory fields in messages that do not use DCCH or CCCH works so that missing mandatory field in information element causes the UE to treat the entire information element as not present. If this happens at the highest level in the message, the UE ignores the message.

Certain test procedures in TS 38.523-1 [12] require messages to include certain fields which differ from standard message contents. As said by Henttonen et al. [10], radio bearers are used in 5G for ensuring a specific Quality of Service at RAN-level. Radio bearers are necessary for data transfer. In order to test establishment of data radio bearer DRB1 for RRC reconfiguration, the specification dictates that *RRCReconfiguration* message must include a radio bearer config DRB1 and an RLC bearer for DRB1. In some test procedures, the specification specifies message contents for both the message starting the procedure and its response. E.g. in UE capability transfer, where the network requests radio access capability information from the UE with *UECapabilityEnquiry* message, which includes *RAT-Type nr*, to which the UE responds with *UECapabilityInformation* message which also specifies *RAT-Type* as *nr*. If modem logs include the complete content of RRC messages, this information can be used to recognize possible issues when analyzing.

### 2.3. Log parsing

Automating modem log analysis requires computers to read the log files. He et al. [13] explain that logs are used in software systems to record different types of runtime information. The developers of the software systems can then use this log information to diagnose problems in production settings. Logs of very complicated systems can easily grow too large for traditional manual inspection, so automated log analysis tools utilizing data mining have been developed. Log messages are often not in a machine-readable form, thus first these unstructured messages have to be transformed to structured data. This process is called log parsing [13]. Log files can be as simple as output of *printf()*-calls saved to a specific file. Their file format can differ, from those simple text files to bespoke formats. A tool designed to analyze log files from different sources must be able to parse these different file formats. Jayathilake [14] mentions the lack of standard for logging as one of the biggest problems with log analysis.

Jayathilake [14] gives expected features for a log parser as a part of a log analysis system. According to them, a log parser should handle log files with various structure and syntax, recognize common tokens in log files, such as timestamps, be able to read from different log sources, and handle possible file corruptions and errors without stopping the parsing. For file formats, they mention text files, XML files, and binary files as formats that a log analysis system should support. Log parsers should store the parsed version of the log, so that time can be saved if the log data is needed again in the future [14].

In order to parse messages properly, their structure must be understood. According to He et al. [13], the typical log message consists of three different fields. Timestamp field records when the event in question occurred. Verbosity level field records the severity level of the event recorded. Message content field includes the message itself, and it can be further split into constant and variable parts. The constant part is always the same for each occurrence of that event. The variable part includes information from the runtime related to the event, such as an IP address. This part can vary for different occurrences of the same event. During log parsing, the parser automatically separates the constant and variable parts of the message and then transforms them into a specific event. Usually, the constant part of the message specifies the event. An event could be specified as a string with asterisks in places where variable parts are place, e.g. "Receiving block \* src: \* dest: \*" [13]. Kc and Gu [15] use a similar method in their log parser, where they refer the constant part as message type. When parsing cellular modem message logs, PDUs of particular protocol messages would be included in variable parts of log messages.

He et al. [13] specify that in traditional log parsing, the parser requires regular expressions to transform messages to specific events. Creating these regular expression rules for modern software systems requires significant effort, and if the system changes over time through updates, these rules have to be updated continuously. Automated log parsing methods can allow the parser to evolve with the system [13].

Fu et al. [16] categorize log messages to five different categories. The first category is assertion-check logging, where Assert or similar functions are used to check if failures happen during execution. These are used to log messages before the execution is terminated. The second category is return-value-check logging, which means that return values of certain functions are checked and logged, if they are unexpected. They note that incorrect return values are used as indicators for potential errors. The third category is exception logging, where the exception context is logged after an exception has happened. This might happen in e.g. Java catch block. The first three categories concern unexpected situations. Unexpected situations can lead to system errors, thus logging them before the error happens is helpful when identifying the error from the log. The remaining two categories are execution points. First execution point category is logic-branch logging, where a code execution path chosen at a branch statement in the code is logged. This means e.g. logging after a switch-statement. Second execution point category is observing-point logging, which includes all logging situations which cannot be categorized in the four previous categories. This might be e.g. recording transactions or other critical events. As a whole, execution points are used to understand code flow and to help when trying to identify the root cause of a failure [16]. In cellular modem log's case, the protocol message PDUs could be thought as either type of execution point log message depending on how the modem is implemented.

Following the survey done by Fu et al. [16], they see log filtering as one possible area for future improvements. Due to large size of files filtering logs with post-processing in order to simplify the process of finding useful information from the logs. Other possible future improvements include better tools for log search, log analysis, and log visualization [16]. A log parser can handle log filtering when parsing the log e.g. by discarding messages which do not fit some predefined criteria.

He et al. [13] provide four different implementations of widely used log parsers through Github. They are easily reusable for both research and practical use. First is Simple Logfile Clustering Tool (SCLT) that works with a three-step procedure. Step 1 is word vocabulary construction during first pass over the data, 2 is cluster candidates construction during second pass over the data, and 3 is log template generation based on the created clusters. Iterative Partitioning Log Mining (IPLoM) is a four-step procedure utilizing heuristics. Step 1 is partitioning the data by message size, 2 is partitioning by word position, with split happening at position where the least amount of unique words appear, 3 is partitioning by search for mapping between unique tokens, and finally step 4 is log template generation similarly to SCLT. Log Key Extraction (LKE) has three steps and uses clustering algorithms and heuristic rules. Its step 1 is log clustering, where messages are clustered with clustering algorithms, then in step 2 clusters split using heuristic rules, and final step is again log template generation. LogSig also works with a three-step procedure, with messages first converted to sets of word-position pairs, then clustering based on those word pairs with multiple iterations, and finally generating log templates [13].

Mariani and Pastore [17] use SCLT to separate event names from their attributes in event logs. Their technique performs the separation in several iterations. On first iteration, only log templates that constitute more than 5% of all events in the log are considered. If log templates are found, the process is repeated for the log events that do not fit the found log templates. If no log templates are found, the threshold is lowered by 25%. These two types of iterations are continued until all events are linked to log templates or the threshold is reduced to 1 event. The use of multiple iterations should help with identifying events, as constant parts of the log messages are more likely to be included in the log templates that satisfy higher thresholds [17].

He et al. [18] introduce Drain, which is an online log parsing method using fixed depth trees. Drain classifies log messages with five steps. The first step is preprocessing, where predefined regular expressions are used on raw log messages to remove commonly used variables, such as identifiers. In step 2, Drain starts to traverse the parse tree by classifying messages by message length. Message length is defined by the amount of tokens present in the message. Step 3 is based on the assumption that earlier tokens of the message are more likely to be constants, and thus better suited for classifying the message. How many tokens Drain uses to classify messages depends on the depth setting. Larger depth setting increases the amount of preceding tokens checked. Classification continues in step 4, with the log message being grouped within the category found in step 3 using token similarity. The final step is updating the parse tree based on the classification of the message. This can either modify an existing classification group, or create a new one. According to evaluation they present, Drain outperforms the other tested log parsers [18].

## 2.4. Failure detection

Once log parser has transformed the log to a more workable form, they can be analyzed with the hope of identifying possible errors and their causes. Jayathilake [14] mention some use cases for log analysis. Software conformance to specification can be verified by comparing log of the software in action to a reference log. System administrators can use logs for system health monitoring and detecting possible unwanted actions, such as system breaches. A log can also be analyzed statistically, determining usage patterns, requirements, and bottlenecks. Anomaly detection is analyzing logs in order to detect problems [14].

In order to analyze different types of logs with differing formats and sources, a log analyzer requires many different features. Jayathilake [14] list expected features for a log analysis system. The requirements for log parser part of the system are described previously in Chapter 2.3. A log analyzer should be able to show the data in a user friendly manner in a user interface. It should provide automation mechanisms for recurring analysis patterns, which allows for a collection of reusable analysis routines. Its user interface should provide information on intrusion detection and standard compliance [14].

Mariani and Pastore [17] classify log file analysis approaches to three different categories: specification-based techniques, expert systems, and heuristic-based techniques. With specification-based techniques, log files are compared to models representing valid event sequences. As the specifications have to be updated manually, it can require a lot of effort by the users of the technique. Expert systems compare events in logs to event patterns that are specified in advance. These event patterns signify system failures, which means that a new pattern has to be specified for each different system failure. Creation and upkeep of these event patterns takes significant amount of effort. Heuristic-based techniques try to avoid the issue of upkeep and specification creation efforts, by applying machine learning to log files. The goal of the learning process is to generate models for accepted logs with little or no human input [17].

Swatch by Hansen and Atkins in 1993 [19] is an example of an older expert system. Swatch watches log files until it encounters an event that matches regular expression provided in Swatch's configuration file. The configuration file also defines how Swatch reacts to noticing that specific log event. The structure of the configuration file is explained more in-depth in Chapter 2.5. Some of the possible reactions to certain log events are echoing the log event to a central terminal where an administrator will see it, sending a bell signal to a central terminal, or mailing the event log to a previously defined list of users. In practical use Swatch was deemed useful. It helped in detecting intruders and prevented system meltdowns when air condition units failed [19].

Jayathilake [14] present Inference Engine for their structured log analysis framework. Inference engine requires the log data to first be parsed by a log parser, which formats the data to a tree data structure. Users provide the Inference Engine with scripts, that it then uses to manipulate the data trees. The Inference Engine outputs a set of trees, which contain inferences made from the data and the scripts. The user interface then can show these trees in a dashboard. The log analysis framework uses SQLite for historical data, because it is lightweight and available for almost all popular platforms. However, in their testing they found that relational databases such as SQLite are not good with message logs that have varying message attributes and nested messages. Instead, NoSQL database that uses varying table schema could be better

[14]. As telecommunication protocol messages often include nested structures, it might be appropriate to avoid relational databases with fixed schemas or extra care should be applied for how those nested structures are stored.

The Inference Engine Jayathilake presents in 2012 [14] uses a mind map based procedural language they presented in 2011 [20]. The language uses mind maps because the way mind maps represent information is more similar to the way human brain does it than alternatives. The mind map is implemented as a tree mentioned above. The language is Turing complete and allows for comments in order to improve usability. It allows for a creation of short scriptlets for automating analysis of recurring patterns in logs. These scriptlets can be used as a part of a larger script without affecting the rest of the script, allowing for different engineers to utilize their knowledge and expertise on log analysis with the same script [20].

Kc and Gu [15] present Efficient Log-based Troubleshooting (ELT) system that is designed for use with cloud computing infrastructures. Their testing results show that it can be used to find previously unnoticed software bugs through log analysis. ELT analyzes log data with a two-step algorithm. First step has the analyzer perform hierarchical clustering using message appearance vector (MAV). After step one, there are several clusters with differing sizes. Small clusters are treated as anomalous clusters, because anomalies are less frequent than normal messages. Step two uses a more precise outlier detection method called message flow graph (MFG). It is used individually for larger clusters created in step one. Using the more fine-grained method only in step two significantly decreases the processing overhead of the analysis process [15].

ELT [15] further simplifies the manual analysis process for the user by extracting key messages from the anomalous clusters. These key messages are supposed to be the most relevant messages for determining the root cause of the anomaly. ELT locates key messages by first comparing anomalies with normal message instances, again using MFG. If certain sequence of messages is present in anomalies but not with normal message instances, the sequence of messages is added to a difference log where the differences are saved. After that clustering is done to the difference logs using MAV. Anomalies that are of the same type should be clustered together. Third step is to compare MFGs of different anomalies of the same type and finding out what message sequences they have in common. Message sequences in the difference logs that match the common sequence are outputted as key messages [15].

ELT [15] can automatically perform invariant checking for the key messages if the invariants have been provided to it beforehand. Performing invariant checking only for the key messages saves execution time. Kc and Gu's example for invariant checking is with Apache's Virtual Computing Lab (VCL). The error they found in the VCL was a *multiprocess forking error*, which violated the invariant "a reservation request cannot be processed by multiple processes simultaneously". They discovered this by checking the start and end timestamps for processes, which made it clear that execution of two processes overlapped [15]. For a 5G cellular modem log, these invariants could be based on the 3GPP specifications.

#### ***2.4.1. Heuristic log file analysis by Mariani and Pastore***

Mariani and Pastore [17] present a heuristic-based log file analysis technique. Their technique does not require specifications to be created and works when failures are caused by multiple separate unexpected events. The technique requires logs of

successful executions. These success-case logs can be collected from real-world use cases or generated, as the technique does not care how they were created or collected. In the technique, models are generated from the success-case logs with three steps: event detection, data transformation and model inference. Once the models have been created, they can be used to compare failures to successful cases. The techniques comparison algorithm can identify both the acceptable and unacceptable event subsequences. The user of the technique can then check through the unacceptable sequences and hopefully identify the root cause of a failure [17].

Mariani and Pastore's [17] event detection requires the log files to be in a specific format before SCLT is used, as described earlier in Chapter 2.3. After event detection, new log files where the templates are included in the log itself are generated. Event detection is followed by data transformation, which replaces variable attribute values with data flow information. They note that attribute data in logs cannot be ignored nor directly compared between different logs when analyzing them. Direct comparison leads to many false positives, where perfectly normal variation in values is flagged as a possible failure. Not taking the data values into account at all causes the analysis to miss important pieces of information. Thus, they created three different strategies for dealing with variable attribute data by replacing them with data flow information. The strategies are global ordering, relative to instantiation, and relative to access. These strategies are applied to log message attributes that work on the same data. The technique works by expecting that correlation exists between attributes that share a certain amount of values. A cluster is created for attributes that correlate with each other. The strategies are applied for clusters individually. Global ordering works by replacing variable values with numbers depending on the order of appearance. This leads to the first value to be replaced with "1", the second with "2" and so on unless the same value was already replaced with a number before. In that case the previously used number is reused. In relative to instantiation strategy variable values are replaced with "0", or if the value was already replaced with a "0" before it is instead replaced with a number that is number new values observed since the value's first occurrence plus "1". This should be helpful in recognizing repeating sequences, where the values change for each separate occurrence. In relative to access strategy variable values are replaced with "0" for their first occurrence and in other cases with a number indicating how many events were in between the current and the last occurrence. This approach captures cases with repeating sequences, but that are missed with relative to instantiation because of reused variable values [17].

Mariani and Pastore [17] implemented a technique for automatically choosing one of the three data flow analysis strategies mentioned previously. This is done by using all three strategies to sets of log message attributes. The strategy that produces the least amount of new symbols is chosen, if it replaces more than 50% of variable values with less symbols. The extra requirements are so that variable values that are present only once in the set do not have too much of an influence on the result. The strategies should not be used if the data flow information requires more than 10 different symbols. In those cases, Mariani and Pastore recommend using only the log templates and not taking variable attribute values into account at all [17].

In Mariani and Pastore's [17] technique after variable attribute values have been replaced with data flow information, they apply kBehavior inference engine to the modified log files. kBehaviour is an inference algorithm that updates the finite-state machine it is creating incrementally [21]. It can update sequences of the finite-state machine when event logs that fit to previously seen patterns are encountered in an

analyzed log. The finite-state machine kBehaviour creates generalizes and summarizes event sequences present in the log. Once a model, which is a finite-state machine, has been generated from the successful log, the technique's final stage can begin. In the failure analysis phase these models are compared with failure logs. The technique aims to find log event sequences from failure logs that differ from the models created from successful logs. When these event sequences are located, the technique's users can inspect them in depth. For a simple way to do the comparison, they present checking if the model compared generates log events (or a trace) found in the log. An anomaly is present if the model does not generate the log events. In some cases the model can generate the log events only to a certain point. In those cases the area around that point should be examined. If a single sequence of log events includes multiple successive anomalies, this method will not find the later anomalies due to it not being possible to match the log with the model after the first anomaly. In addition to the simple method, Mariani and Pastore also developed a matching process that can be used to recognize event sequences from the log no matter where they are located. They compare event sequences with different sections of the model, and in case of matches add them as possible extensions to the model. The users then have to inspect these extensions. This solves the problem of not being able to compare later log events with the model if an anomaly is found [17].

In order to test their technique, Mariani and Pastore [17] analyzed three different applications. For two of the applications, only one failure case was studied, but for the third application analysis was done for three different cases. They present the percentage of suspicious events, amount of false positives, the number of true positives and the precision of results for all five study cases. The technique cut down the amount of events the users had to analyze manually in all cases. They had a moderately high amount of false positives, which they blamed on incomplete samples used when generating the models and limited generalization during it. Some true positives were only found because of also analyzing data flow information, which demonstrates that analyzing variable attribute values is important [17].

## 2.5. Metasyntax and data formatting

A log analyzer needs a set of rules inputted in order to know what to look for in the logs. These rules can define for example what log messages and message sequences are acceptable, or what log messages or message sequences signify failures. The rules can be defined by the users of the analyzer, or automatically based on features of the source code or previous, usually successful, logs, as is the case in the article of Mariani and Pastore [17].

Log parsers also need information on how to read the logs. The log format can be provided in the same configuration as the analysis rules. Format for each specific message can be defined with regular expressions, or the log parser can derive it automatically. In the latter case, there is no need to provide the format of specific messages, as the format of the log should be enough.

For Swatch [19], one configuration file provides all the necessary information for both log parsing and analysis. The configuration file defines a pattern expression, action for that expression, time interval used for discarding redundant messages, and location of time stamp in the log message. Each field is separated with a tab. Line breaks separate different sets of fields. Pattern and action fields can optionally include more than one value. This can be used to configure the same reaction to multiple



different log messages, or many reactions to the same message with a single line in the configuration file. Values are separated with commas. The time interval field is optional and its default unit is seconds. It can optionally include minutes and hours also. The location of time stamp is also optional and it can only be used if time interval field is present. It consists of the start index of the time stamp and its length. The configuration file is read from top to bottom, with earlier lines having precedence in cases when a log message matches with multiple different patterns. The configuration file supports comment lines, which are created by starting a line with “#” character [19]. An example of a valid line of text for the configuration file can be seen in Figure 8.

|                  |      |    |      |
|------------------|------|----|------|
| Message received | echo | 20 | 0:24 |
|------------------|------|----|------|

Figure 8: An example of valid line of configuration for Swatch

Rules of communication for computer science can be defined with metasyntax notations. This includes rules of telecommunication protocols. Sellink and Verhoef [22] present some use cases for descriptions made with metasyntax notations for programming languages. They can be used as guides for compiler implementation, they can be used as manuals, and they can be used when implementing reengineering tools for the language described. Sellink and Verhoef focus on the reengineering use case, where for example language descriptions can be used to generate a full documentation for the language. They present some methods on how to extract language descriptions from electronic language manuals. The manual they analyze violates certain conventions it itself establishes, which complicates machine-reading it [22].

One of the most commonly known metasyntax notation is Backus-Naur Form (BNF) and its many variants. Backus [23] described the original “metalinguistic formula” over 60 years ago in 1959 and used it to describe ALGOL syntax. The original Backus-Naur form has strings inside “ $\langle \rangle$ ” symbols. These strings represent variables of the formula. Two metalinguistic connectives used are “:=” and “or”, signifying definition of a variable and alternate definitions for those variables. All other symbols are then symbols of the language being described. BNF supports recursive definitions, meaning that for example variable  $\langle \text{String} \rangle$  can be recursively defined to be constructed of smaller strings. Backus et al. [24] describe a different variant of the formula, where “or” is replaced with “|”-symbol. Originally the formula was called Backus Normal Form. Knuth [25] proposed instead using the name “Backus Naur Form”, as the formula is not a conventional “Normal Form” and the same concept is known with different names in linguistics. “Backus Naur Form” also takes into account Naur’s additions and influence in the formula being adapted more widely. The abbreviation BNF works for both names.

BNF has been expanded several times and adapted for different purposes, for example ISO/IEC 14977:1996 standard for Extended BNF [1]. EBNF includes many common extensions to original BNF, such as: quoting terminal symbols (smallest possible symbols of the language that cannot be split further) which allows for any character to be used as a terminal symbol of the defined language, “and” for optional symbols and repetition, explicit final characters for rules, and support for brackets for grouping. EBNF also includes other new features which are based on experience with

formal definitions: defining an explicit amount of items, such as a field with specific length; defining few exceptional cases; defining comments for the language; defining names of symbols in the language with several words; and support for later extensions. EBNF is designed for use with languages with simple grammars and it is not suitable for defining grammars that are more complex. This was done because EBNF's main need was to be suitable for more common use cases. An example for EBNF provided by the standard can be seen in Figure 9. The example shows how consonants are defined from letters and vowels with an “except-symbol” (minus-sign) [1].

```
letter = "A" | "B" | "C" | "D" | "E" | "F"
      | "G" | "H" | "I" | "J" | "K" | "L" | "M"
      | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
      | "U" | "V" | "W" | "X" | "Y" | "Z";
vowel = "A" | "E" | "I" | "O" | "U";
consonant = letter - vowel;
```

Figure 9: One example of EBNF from the standard [25]

Different notations can be used to describe message syntaxes. Dubuisson [5, pp. 487-494] lists some of them. ASN.1 is presented in Chapter 2.5.1. XDR (eXternal Data Representation) notation is designed for remote procedure call communication between a client and a server. XDR does not fit well to complex structures due to certain missing features, such as no support for optional components in structures. NIDL (Network Interface Description Language) notation resembles XDR notation, but it is used with a different encoding method, where the sender includes its own architectural information in the message that the receiver can then use to decode the message. This leads to a need for several different converters if the receiver receives messages from heterogeneous devices. IDL language for CORBA (Interface Definition Language and Common Object Request Broker Architecture respectively) is a language used to describe interfaces for classes and services in object-oriented CORBA architectures. It resembles ASN.1 and translation between the two notations are relatively simple, but some information is lost when translating from ASN.1 to IDL. IDL does not support generic integer types, forcing specification creators to use a pre-defined integer type. RFC 822<sup>4</sup> notation is a simple way to exchange data for heterogeneous systems, but it is also bandwidth expensive. It is based on describing the data in an alphabet readable by both systems. EDIFACT (Electronic Data Interchange for Finance, Administration, Commerce, and Transport) notation is a graphical notation and encoding rules for messages that were designed to replace previous physical documents. It is not very flexible or extendable, as the encoding is position-based and all the messages have to be standardized [5].

### 2.5.1. ASN.1

As ASN.1 is used in 3GPP's 5G standards, understanding how the messages use ASN.1 and how it can be decoded is important for analyzing message contents. Dubuisson [5, pp. 60-63] describes the history of ASN.1. ASN.1 can be traced back to

<sup>4</sup> <https://tools.ietf.org/html/rfc822>

*Courier* notation, which was used to represent Remote Procedure Call data. The notation was adapted and standardized as X.409 by the Consultative Committee on International Telephony and Telegraphy (CCITT) in 1984, which was designed for use with email Message Handling Systems protocols, but could be used independently from it. This was not necessarily by design, as needs of email protocol were arbitrarily complex. This led to the notation being adapted by industry and engineers working on different levels of the OSI model. ASN.1 was then created in 1987 using X.409 as a basis. ASN.1 is technically equivalent to abstract notation part of X.409, but its documents were fully rewritten to take into account the presentation layer of ISO model. In 1989, CCITT published X.208 document for ASN.1, which replaced the previous X.409, and in 1995 the standard called ASN.1:1994 was published in four parts (X.680, X.681, X.682 and X.683) [5]. The latest version of the standard was released in 2015 <sup>5</sup>.

Dubuisson presents an indepth User's Guide and Reference Manual for ASN.1 [5, pp. 95-98]. The User's Guide acts as a guide for someone who is beginner when it comes to ASN.1. The Reference Manual is meant for experienced users. It is supposed to be used as a reference for particular details about ASN.1. ASN.1 grammar rules are expressed with EBNF in the Reference Manual. Thus ASN.1 type BOOLEAN is described in EBNF with "BooleanValue  $\rightarrow$  TRUE | FALSE" [5, p. 129].

According to Dubuisson [5, pp. 463-467], the most important tool in protocol implementation is the compiler. For ASN.1, the compiler can generate encoding and decoding procedures for the data types defined in the specification given to it. Usually, an ASN.1 compiler reads ASN.1 modules that are linked with imports, and outputs C, C++, or Java code. Ideal ASN.1 compiler works in 4 stages: lexical analysis, parsing, semantic analysis, and code generation. The compiler should only start working on the next stage if the previous stage was completed without errors. Errors in the earlier stages can lead to errors in the later stages, but error messages of the later stages can in those cases be useless. Errors during lexical analysis and parsing are usually caused by symbols or grammatical structures forbidden in ASN.1 grammar. Errors during semantic analysis are instead caused by incoherence in the input specification. If no errors happen, the final step is the compiler generating encoding and decoding procedures with the data types specified in the ASN.1 specification for the target language. If the target language is C, it often means a header file with the data types, and a source code file for the procedures. These files can then be given to another compiler (for example a C compiler) which generates the final executable [5].

Dubuisson [5, pp. 469-470] explains that ASN.1 grammar has some features that make it inherently difficult to parse. An example of such feature is the lack of a semicolon at the end of a definition. This is because ASN.1 was originally created for communication between a standardization committee and application designers, meaning that deriving encoding and decoding procedures directly from the specification was not a planned use case. This is demonstrated by analyzing ASN.1:1997 standard grammar with common parser generators (Yacc, ANTLR). This leads to the tools issuing hundreds of conflicts. The way to avoid this issue is to transform the standard to a new grammar that is easier to parse. An example of easier to parse version of ASN.1 grammar would be LL(1)-compliant versions, which Dubuisson used as preliminary works for their own Asnp-parser [5].

---

<sup>5</sup> <https://www.itu.int/rec/T-REC-X.680/>

## 2.6. Summary

From this background research, assumptions about what type of data would be relevant to find in cellular modem logs can be made. 3GPP specifications define the functionalities of cellular modem layers. Cellular modems use primitives and PDUs for communicating within and between layers. For 5G NR RRC specifically, TS 38.331 [10] is the primary source for how the RRC protocol works and what kind of operations could be detected in the logs.

There already exists log parsers and analysis methods of various types. Log parsers are used to transform log message information to structured data. Exact features of log parser depend on the type of log being parsed, but automatic generic parsers also exist. For Protocol Log Analyzer, the log parser has to work with the likely proprietary log format used by the modem. Already existing analysis methods do not exactly fit with the needs of Protocol Log Analyzer as they lack the features for checking logs for event sequences that follow complex predefined patterns, but lessons can still be learned on how to develop and use the Protocol Log Analyzer. These include how the analysis method should take attribute values into account to improve accuracy and how an anomaly can be detected if the log does not include expected event sequences [17]. Analysis methods that utilize automated heuristics could be useful in cellular modem analysis, but implementing and testing them is outside of the scope of this work.

Protocol Log Analyzer requires some way to define what to look for in the logs. Swatch [19] configuration file works as a simple example on how the events of protocol log could be provided for the it, with pattern and the reaction to that pattern defined separately. Metasyntax notations, such as BNF [23], provide tools for defining more complex event sequences, consisting of several log messages. Notation that integrates Swatch style configuration files with more complex grammar should be developed for Protocol Log Analyzer. It should combine that new notation with an analysis method that supports checking for events that can be described with the notation. The log parser's output must be understandable for the analysis method implementation, so some type of interface has to be designed. As a whole, Protocol Log Analyzer should inform the user on log message sequences matching with the written rules found during analysis.

### 3. REQUIREMENTS

This Chapter defines the requirements for Protocol Log Analyzer, which engineers analyzing modem logs could use to ease their workload. Engineers with industry experience in designing and implementing RRC protocol for cellular modems were consulted for advice and proposals. The consulting was organized as a group discussion at Mediatek Wireless Finland Oulu office in May of 2021, where the engineers were presented with the idea of a protocol log analysis tool (Protocol Log Analyzer) and could bring up their own ideas, wishes, and expectations for it. Notes written about the group discussion were later organized and used as a basis for the requirements.

As some requirements are derived directly from wishes of the engineers, others are derived from features required for any type of log analyzer, based on previous understanding of software development and expected features of a log analysis system described by Jayathilake [14]. Wishes of the engineers are scoped into requirements with the perspective that the goal is to create a proof-of-concept tool for the techniques used, instead of a software product ready for their everyday use. This meant that wishes related to integration with specific systems are not included in the requirements.

The requirements are split into three sections, starting with the highest-level or most abstract requirements called utility requirements. Analysis and technical requirements are more directly related to design and implementation of Protocol Log Analyzer, referred as the tool in this Chapter due to requirements being defined for a generic protocol log analysis tool.

#### 3.1. Utility requirements

These requirements concern what the user, meaning the engineers, must be able to achieve using the tool, or what utility the tool must provide. Utility requirements are listed in Table 1. Important requirement is that the tool should help its users perform log analysis faster and more precisely than going through the logs manually (REQ\_1). Without tangible time, efficiency, accuracy, and/or usability benefits the users will not adopt the tool to their daily usage. For this, the user experience must be good enough, meaning the tool must include a user interface (UI). The UI should be graphical (REQ\_2) in order to better fulfill REQ\_1.

Fundamentally, a log analyzer must work with logs. One of the requirements is thus that the tool takes a log as input and outputs analysis based on predefined rules (REQ\_3). The users must be able to define these analysis rules themselves for their specific needs (REQ\_4), similarly to how configuration files work for Swatch [19]. The analysis rules are used to define different events that the tool looks for within the inputted log. Examples of defined events are a specific protocol message, which includes a PDU with a specific field, or specific message (*RRCReconfigurationComplete*) appearing in the log as a response to a previous message (*RRCReconfiguration*). Because the tool is primarily meant for cellular modem logs, the tool must understand 3GPP protocol messages found in the logs (REQ\_5).

Table 1. Utility requirements

| Name  | Definition                                                                                                                                   |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------|
| REQ_1 | The tool shall provide tangible time, efficiency, accuracy, and/or usability benefits for its users compared to analyzing the logs manually. |
| REQ_2 | The tool includes a graphical UI.                                                                                                            |
| REQ_3 | The tool takes a log as input and outputs analysis based on predefined rules.                                                                |
| REQ_4 | The tool allows the users to define the used analysis rules.                                                                                 |
| REQ_5 | The tool can read and analyze 3GPP protocol messages from the logs.                                                                          |

### 3.2. Analysis requirements

Requirements for analysis features are the features the tool's users need for effective analysis. They are related to utility requirements REQ\_4 and REQ\_5. These requirements define what features the rule notation used in the tool must support and what type of analysis information it must provide to the users. Analysis requirements are listed in Table 2.

The rule notation requirements (REQ\_6 to REQ\_11) are formed with the assumption that rule notation is an integral part of this work and that evaluation includes checking if it can be used to recognize different kinds of issues from logs. This means that its requirements for proof-of-concept stage are essentially identical to what is required from a complete product.

REQ\_12 is there to make sure the tool can link found events back to specific locations in the log. This is an important feature for log analysis, as it allows the user to instantly locate objects of interest from the log.

Table 2. Analysis requirements

| Name   | Definition                                                                                                                   |
|--------|------------------------------------------------------------------------------------------------------------------------------|
| REQ_6  | The rule notation allows for defining expected log message order.                                                            |
| REQ_7  | The rule notation allows for defining how the tool reacts to specific events in the log.                                     |
| REQ_8  | The rule notation allows for defining both "X shall not if Y" and "X must if Y"-type rules.                                  |
| REQ_9  | The rule notation allows for defining new rules that include previously defined events within the new larger event.          |
| REQ_10 | The rule notation allows for defining complex log message content, including PDU structures.                                 |
| REQ_11 | The rule notation allows for defining expected timescale of messages within an event.                                        |
| REQ_12 | The tool communicates the result of the analysis to the user, including where the log's content does not match the expected. |

### 3.3. Technical requirements

Technical requirements concern which features are needed from the implementation so that the tool can reach analysis and utility requirements. These are the requirements most closely related to design and implementation of the tool. The technical requirements are listed in Table 3.

Derived from REQ\_3, the tool must read and parse log files (REQ\_13). These logs must then be transformed to an internal representation model, which the tool uses during the analysis process. The internal representation model must work with the analysis requirements, see REQ\_6 to REQ\_12, which leads to REQ\_14, REQ\_15, and REQ\_16.

According to REQ\_1, using the tool should be simple in daily use cases engineers encounter. As the defined analysis rules are an important part for the utility of the tool, they should be easily readable and modifiable. This means that the rule notation should guide towards writing understandable rules. Those rule-files should be small so that users can share their creations easily. The rule-file format should be common enough, that the users can open, modify, or create new ones with standard text editing programs, such as Microsoft Notepad.

Table 3. Technical requirements

| Name   | Definition                                                                                        |
|--------|---------------------------------------------------------------------------------------------------|
| REQ_13 | The tool parses logs to an internal representation model.                                         |
| REQ_14 | The internal representation model retains message order and timestamps.                           |
| REQ_15 | The tool can link a message in the internal representation back to a specific message in the log. |
| REQ_16 | The internal representation model retains message contents, including PDUs.                       |
| REQ_17 | Rule-files are easy to create, read and modify with common programs.                              |
| REQ_18 | The rule notation encourages writing easy-to-read-files.                                          |
| REQ_19 | Rule-files are small for easy distribution.                                                       |

## 4. DESIGN

This Chapter presents the design of Protocol Log Analyzer and a rule notation for it. The design aims for it to fulfill the requirements presented in the previous Chapter regarding the type of analysis it is able to do, the technical requirements needed for those features, and the type of utility the final version has to provide its user.

### 4.1. Architecture

Protocol Log Analyzer is designed with four different modules or sections:

- log parser
- analyzer
- the rule notation and its reader
- UI

These different modules can then be developed and changed separately. This split is not necessitated by the requirements, but is instead made based on previous experiences. The parser, the analyzer, the rule reader, and the UI are connected through interfaces. The parser takes the log file being analyzed as input and outputs the same log data in internal representation model form. The rule reader takes the rule file as input and outputs analysis rules in a format the analyzer can understand. Output of the parser and rule reader are the input for the analyzer. The analyzer outputs results, which are then input for the UI. Flow of data is visualized in Figure 10.

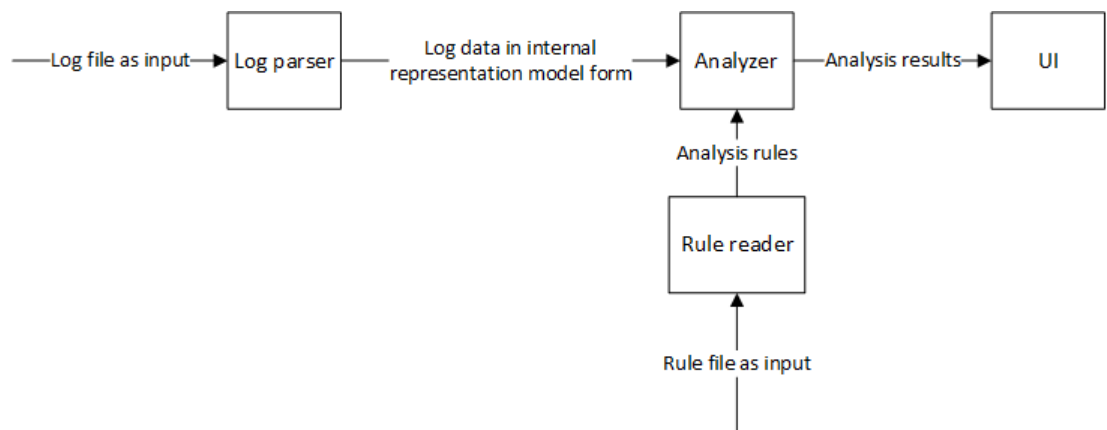


Figure 10: Flow of data between the modules

Protocol Log Analyzer uses internal representation model when applying analysis rules to the log data. REQ\_10, REQ\_12, REQ\_14, REQ\_15 and REQ\_16 requirements are the base for the four fields for each log message in the internal representation:

- index, the position of the log message in the log
- timestamp, directly from the log
- message, “name” of the log message
- content, rest of the content of the log message after the other three fields have been extracted



During analysis, Protocol Log Analyzer has to compare several log messages to the analysis rules. The entire log is not in memory (in internal representation form) due to possible memory issues when there are thousands of messages.

## 4.2. Log parser

The log parser is responsible for transforming log messages to the internal representation model. Log parser must be configured for specific logs, so that it can extract specific fields with high enough accuracy for analysis. The log parser outputs a database file that acts as input for the analyzer.

The log parser works message by message. How different log messages are separated in the log file varies. The design does not commit to any specific type of log file, but it does not include an option for separate configuration file that would allow the same parser to be used for different kinds of logs. The log parser handles each message and builds a database table where each row corresponds to a specific message in the log. The columns of the table are the same as the four fields of the internal representation model: index, timestamp, message, content.

The operation flow, as can be seen in Figure 11, of the log parser starts with the creation of a database. The database is a relational database [26], with data stored in tables formed of columns and rows. This database is used to store the log data in a format closer to the internal representation model. After the creation of the database, the log parser handles the first message found in the log. It extracts information required for the analysis from the log message. How exactly the index and the timestamp are located in the log message, how the log parser names the log message, and what exactly are included in the content depends on the log. The extracted information is formatted to row of the database table. How the data of certain log messages is stored in the 'content'-column has to be done so that the analyzer can handle contents of PDUs and other similar structures. At this point, if there are still unhandled log messages in the input log, the log parser starts handling the next one. The loop finishes when all the log messages from the log have been handled. Then the log parser finishes the creation of the database, which can include committing it to the disk.

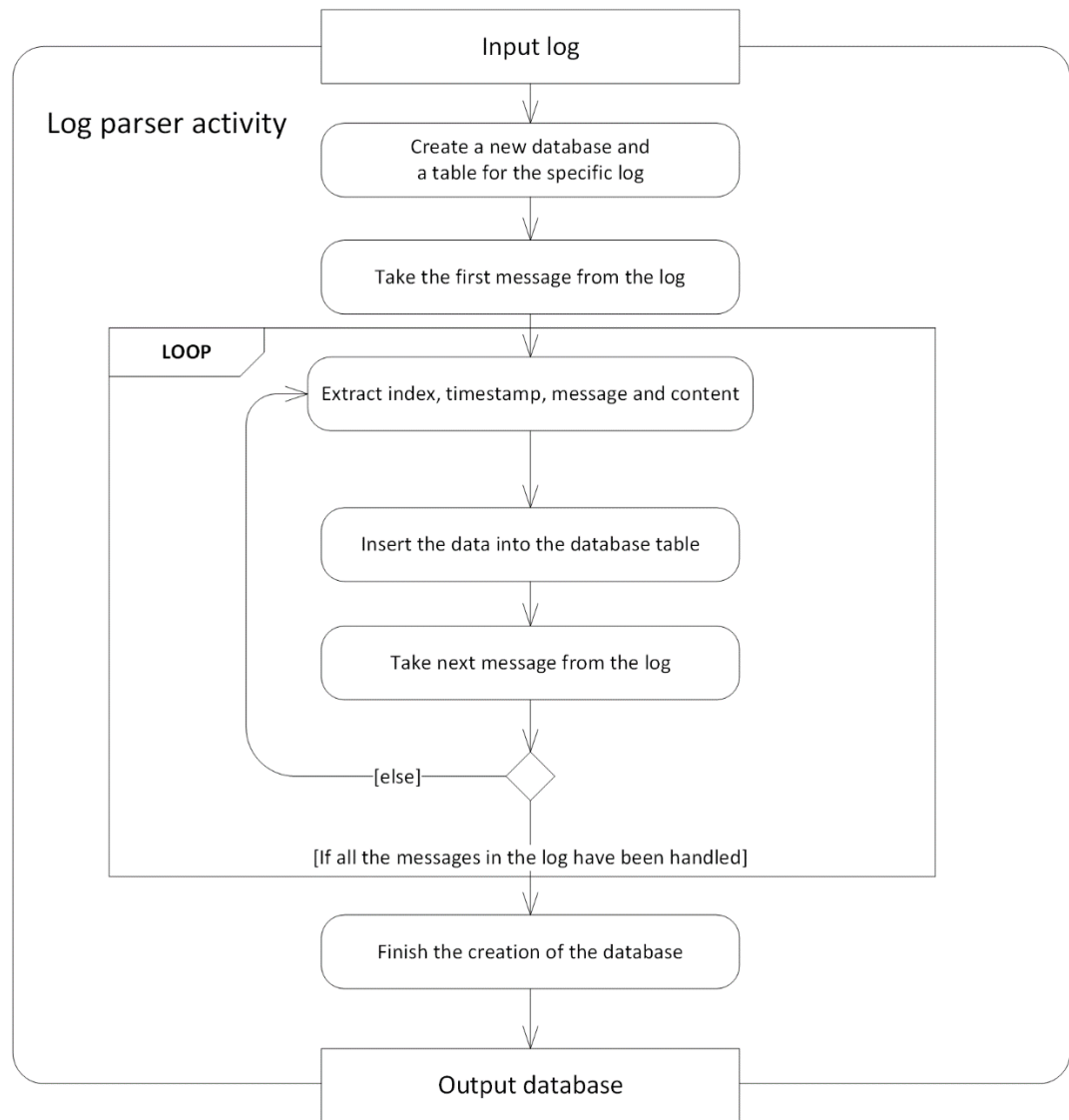


Figure 11: Log parser inner operation flow

### 4.3. Rule notation

The user of Protocol Log Analyzer uses the rule notation for defining the analysis rules. When defining an event with the rule notation, the user has to name it, define the contents, and define how Protocol Log Analyzer should react to it. The end user does not have to be the same person as the user writing the rules, but Protocol Log Analyzer does not include any rules by default. Different types of logs most likely require different rule files, but properly made rule sets should be reusable for different logs of the same type. The rules are read left-to-right, top-to-bottom. The events can have arbitrary names consisting of letters (both uppercase and lowercase letters), digits and underscores. The different possible reactions for Protocol Log Analyzer are defined in the rule notation, thus when writing the rules the user chooses the best suited reaction. The user can define expectations for appearance and content of specific events and log messages. The rule notation relates mostly to requirements REQ\_6 - REQ11. The user can use whitespace characters to format the rules to be easily readable by other users, following REQ\_18.

The rule notation uses certain symbols for specific actions, called operators. These operators cannot be used in event names. The user uses operators to define the rules. The operators used were inspired by EBNF [1] and the C-language [27]. The operators can be seen in Table 4.

Reserved keywords are another tool for the user. They are used for another set of predefined operations in conjunction with the reserved symbols. Additional arguments for the keywords are given within the parentheses, with optional arguments contained within circle brackets. List of keywords and examples for each of them can be seen in Table 5.

Table 4. Operators of the rule notation

| Operator | Usage                                                                                 |
|----------|---------------------------------------------------------------------------------------|
| =        | Definition, 'IS'                                                                      |
| ,        | Concatenate, separation within event definition, defines order of events and messages |
| ;        | Termination                                                                           |
|          | 'OR'                                                                                  |
| &        | 'AND'                                                                                 |
| !        | 'NOT'                                                                                 |
| { }      | Event group, used to define borders of events                                         |
| ( )      | Grouping, used to define groups within events                                         |
| "        | Used to define strings                                                                |
| ""       | Used to define strings                                                                |
| /* */    | Comment                                                                               |
| ==       | Left and right values are equal                                                       |
| >        | Left value is larger than right value                                                 |
| <        | Right value is larger than left value                                                 |
| >=       | Left value is larger or equal to right value                                          |
| <=       | Right value is larger or equal to left value                                          |
| .        | Used to specify nesting in PDU-value definitions.                                     |

Table 5. Keywords of the rule notation

| Keyword                                                           | Usage                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Example                                                                                                                                                                                   |
|-------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Event                                                             | Used to signal the start of event definition. The definitions ends at termination “;”.                                                                                                                                                                                                                                                                                                                                                                                                                                         | Event event0 =<br>{Message(“msg1”)};                                                                                                                                                      |
| Message(name,<br>[other keywords or<br>expected content])         | Used to define a message that appears in an event. Name can be any string. Other parameters are either strings or other keywords specifying content or features of the message.                                                                                                                                                                                                                                                                                                                                                | See other examples.                                                                                                                                                                       |
| Maxdelay(time,<br>[previous_msg])                                 | Used when setting the maximum time delta between different messages. Is attached to the later message in the rule set. Uses the same unit of time as the log.                                                                                                                                                                                                                                                                                                                                                                  | Event event1 =<br>{event0,<br>Message(“msg2”,<br>Maxdelay(200,<br>previous_msg=“msg1”))<br>};                                                                                             |
| Mindelay(time,<br>[previous_msg])                                 | Used when setting the minimum time delta between different messages. Is attached to the later message in the rule set. Uses the same unit of time as the log.                                                                                                                                                                                                                                                                                                                                                                  | Event event1 =<br>{event0,<br>Message(“msg2”,<br>Mindelay(200,<br>previous_msg=“msg1”))<br>};                                                                                             |
| Pdu(start_index,<br>PDUcontent<br>**[PDUcontent],<br>[end_index]) | Used to define where PDUs in log message content are located and what they are expected to contain. There can be an arbitrary amount of different expected PDU contents, which are separated with concatenation symbol “;”. Start and end indices are used define where the PDU can be found in the message’s content-field in cases where the PDU is not the entirety of the message’s content. The analyzer splits the message’s content from the indices, and transforms the content between them to an appropriate format. | Event event2 = {<br>Message(“msg1”, Pdu(0,<br>(PDU.rrc_TransactionId<br>entifier == 5),<br>PDU.criticalExtensions.r<br>rcSetupComplete.<br>nonCriticalExtension,<br>end_index=128)<br>}); |

As can be seen in TS 38.331 [11], PDUs are defined with ASN.1 in 3GPP specifications. ASN.1 would however be too wordy for the rule notation. Instead, the user should only define the expected value for the specific field they want checked. This is done by specifying the location the same way a specific field is specified in nested C-language structs, with dot(.)-operator separating the name of the member and the variable [27]. There is no upper limit for the amount of nesting allowed in the analyzed PDUs. If the users want to make sure a PDU includes a specific optional field, they do not have to set any specific value for it in the rule set. Examples can be seen in Table 6.

Table 6. PDU content examples

| Example                                                      | Definition                                                                                                          |
|--------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| PDU.rrc_TransactionIdentifier == 5                           | Field ‘rrc_TransactionIdentifier’ is set to 5 in the PDU.                                                           |
| PDU.criticalExtensions.rrcSetupComplete.nonCriticalExtension | Field ‘nonCriticalExtension’ exists within ‘rrcSetupComplete’, which exists within ‘criticalExtensions’ in the PDU. |

#### 4.3.1. Event structure in the notation

Events are defined with “=”-symbol. Event-name is on the left, the definition on the right. The right side contains the expected messages, other information related to the messages, and the reaction type if the event is observed in the log. At least one message has to be specified for a valid event definition, but there is no upper limit for the length of an event. Events defined previously in the file can be reused as sub-events in place of messages by using the names of those events. Figure 12 presents an example of how an event is defined from multiple messages and how the reaction is specified. The message in question can be read in English as follows: If “msg2” does not follow “msg1” within 100 time units, alert.

```
/* Example event */
Event event_name =
{
    Message("msg1"),
    !(Message("msg2", Maxdelay(100))),
    notify="Alert"
};
```

Figure 12: Example event definition

#### 4.3.2. Specifying reactions

The users can specify different type of reactions for each event. These are used as instructions for the analyzer and the UI on how to present information found in the logs for the user. If the reaction type is not specified, default reaction (Alert) is used. The reaction type is specified with “notify”-optional argument for Events. The three different types of reactions are explained in Table 7.

Table 7. Types of reactions

| Reaction type | Use case                                                                                                            |
|---------------|---------------------------------------------------------------------------------------------------------------------|
| Alert         | Standard reaction. The UI informs the user with a notification. Used for possible errors and other uncommon events. |
| Visualize     | The UI visualizes the area in which the event happens. Used for common events of interest, such as RRC connections. |
| None          | The UI does not inform the user at all. Used when defining events which are used as sub-events.                     |

#### 4.3.3. Chaining different operators and keywords

In order to provide the users with enough options on how they can define events, the rule notation allows for chaining different symbols and keywords within event definitions. Parentheses (‘()’) are used to group these appropriately.

Figure 13 shows 4 examples of different ways symbols and keywords can be used together. Event 1 has maximum delay applied to ‘msg2’, but not ‘msg3’, and the event is recognized if either message happens. Event 2 requires ‘msg2’ to follow ‘msg1’ and there to be no ‘msg3’ during the next 150 time units after ‘msg1’. ‘msg3’ appearing within that time invalidates the event. Event 3 utilizes ‘&’ for an event where both ‘msg2’ and ‘msg3’ appear after ‘msg1’, but their order does not matter. In Event 4,

there is both a maximum and a minimum delay after ‘msg1’ before ‘msg2’ should happen for the event to be recognized.

```

/* Example events with more symbols */
Event event_1 =
{
    Message("msg1"),
    Message("msg2", Maxdelay(100)) | Message("msg3") /* msg2 within 100 time units of
                                                         msg1, or msg3 */
};

Event event_2 =
{
    Message("msg1"),
    Message("msg2") & !(Message("msg3", Maxdelay(150))) /* msg2 and no msg3 within
150 time units*/
};

Event event_3 =
{
    Message("msg1"),
    Message("msg2") & Message("msg3") /* msg2 and msg3, order does not matter */
};

Event event_4 =
{
    Message("msg1"),
    Message("msg2", Maxdelay(200), Mindelay(100)) /* msg2 comes after msg1 with
                                                         a delay of 100-200 time units */
};

```

Figure 13: Example events using different operators and keywords

#### 4.4. Rule reader

Similarly to the log, the rule file has to be parsed before it can be used in the analysis. This is done by the rule reader, which reads the rule file from top to bottom, event by event. Due to this approach, events defined later in the rule file can use previously defined as sub-events, but earlier events cannot use the later defined events. The rule reader outputs analysis rules to the analyzer. How the output is transferred to the analyzer is not part of the design and is left to the implementation.

Because the rule notation is only used for this specific purpose, its reader does not require all the features of a full-fledged compiler or an interpreter. The reader works on the assumption that the writer of the rules writes them following a specific style and does not try to write rules that lead to unexpected consequences. Errors arising from unintended interpretation of rule notation syntax and examples are accepted as possible.

In Figure 14 the high level logic of the rule reader can be seen. The rule reader recognizes variable definitions and event definitions from the rule file by iterating

through it character by character until it can build words that are either variable names or the keyword 'Event'. The rule reader remembers variable definitions so that it can later replace the variables with appropriate values. Definitions begin with either of those two options. Rule reader has finished its task when it reaches the end of the rule file. Handling of non-valid sequences or syntax is not included in the design.

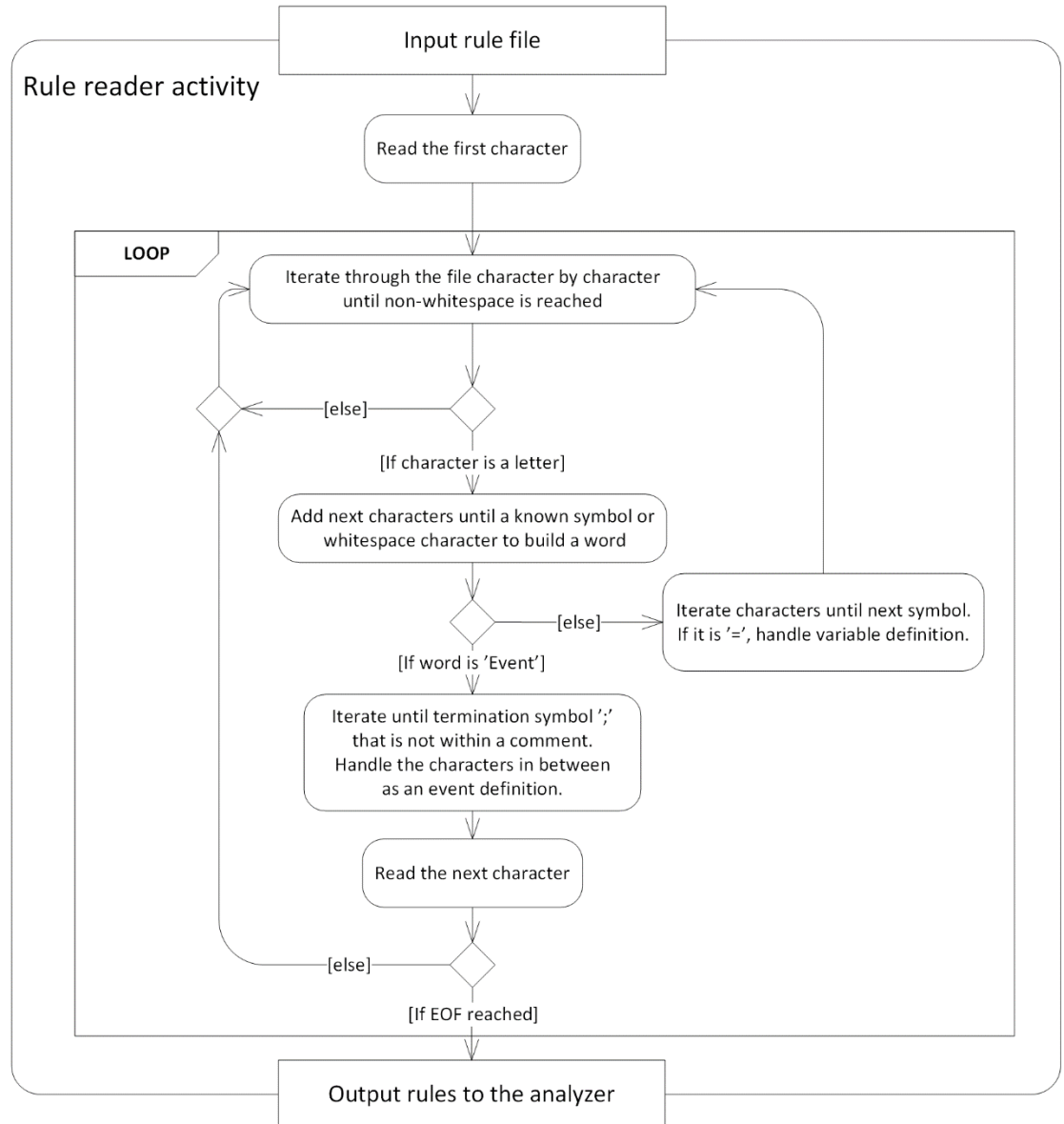


Figure 14: High-level flow of rule reader

Another loop exists within the high-level operation flow specifically for handling individual event definitions. Event definition handling is visualized in Figure 15. Left hand side of '=' is handled first, as it gives the event its name. As the contents of an event are defined within curly brackets, the loop focuses on parsing contents within them. Whitespace characters are ignored. Characters that are valid in event names are used to build words. These words must match with 'Message'-keyword or with the names of previously defined events, otherwise they are not recognized as valid and error handling starts. Operator handling happens when they are encountered. Certain operators affect messages that were already inserted into the event definition (e.g. '|')

OR marking the previous message as optional), others only affect parts of the definition that are not yet handled.

Messages and sub-events are handled similarly, but in sub-events' case the rule reader must retrieve the previously defined event's contents and insert them to the current event. As the 'notify' denotes the parameter for reactions, it is set for the event appropriately when encountered. When everything up to the closing curly bracket has been handled, the loop ends and the rule reader continues with the next definition.

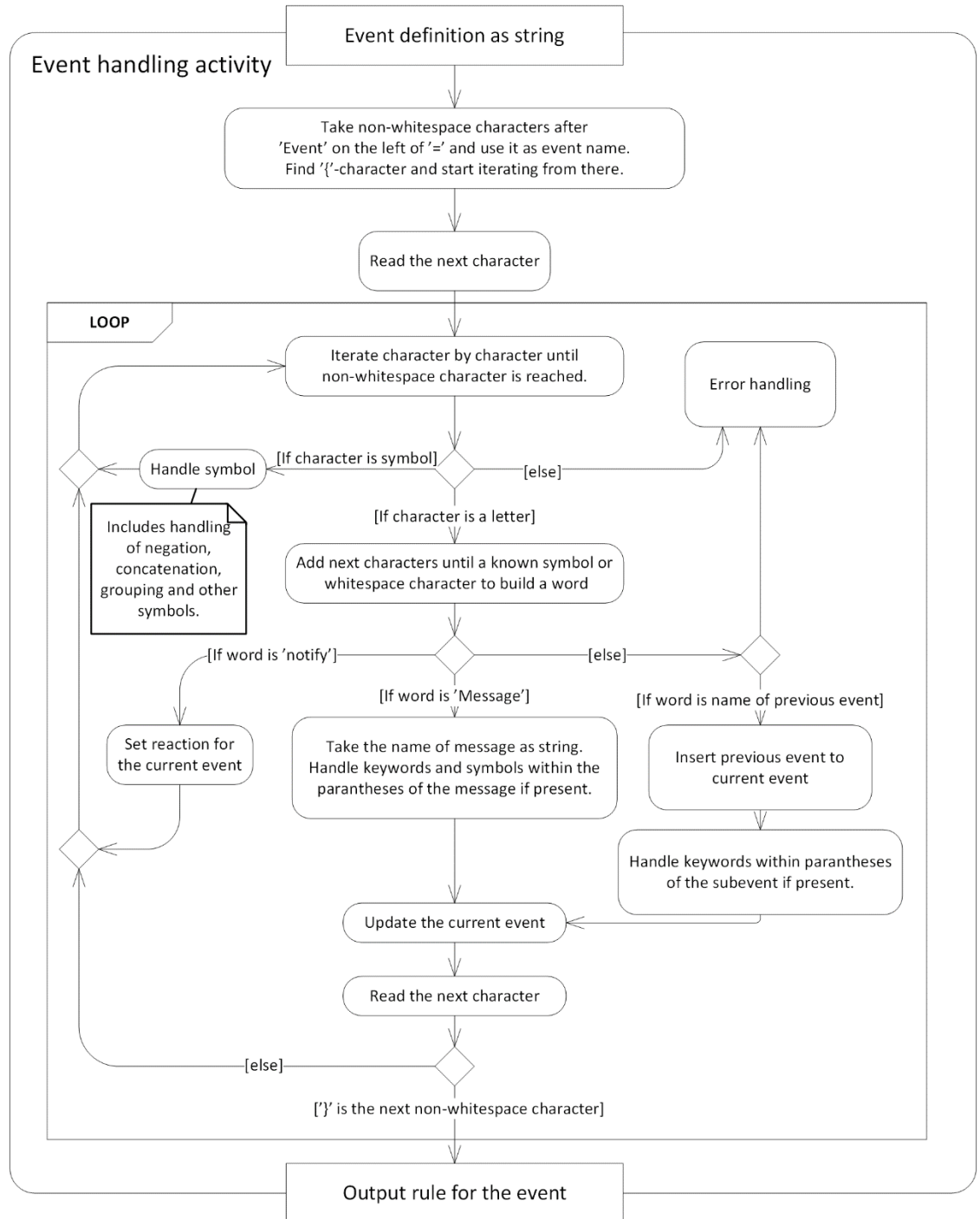


Figure 15: Event definition handling



#### 4.5. Analyzer

The analyzer is responsible for applying the rules to the log data. It takes two different inputs, the database that includes all the log messages from the log parser and the analysis rules from the rule reader. The design for the internal operation logic of the analyzer can be seen in Figure 16.

The analyzer goes through the database in rising index order, matching the order of the messages in the original log. The analysis works by checking if a message's name matches with the next expected message of an event as defined by the rule set. As there can be several occurrences of an event overlapping in the log data, the analyzer compares the names of all possible next messages. This is done by creating an 'ongoing' event whenever a first message of an event is matched with the log data.

After a message has been checked against the rules, the analyzer checks if all ongoing events are still valid. In case the event includes rules that would cause it to be invalid, it is removed from ongoing events. If a complete event has been located from the log data, its name, type of reaction, and indices of the messages are added to 'found' events. These found events are the output of the analyzer.

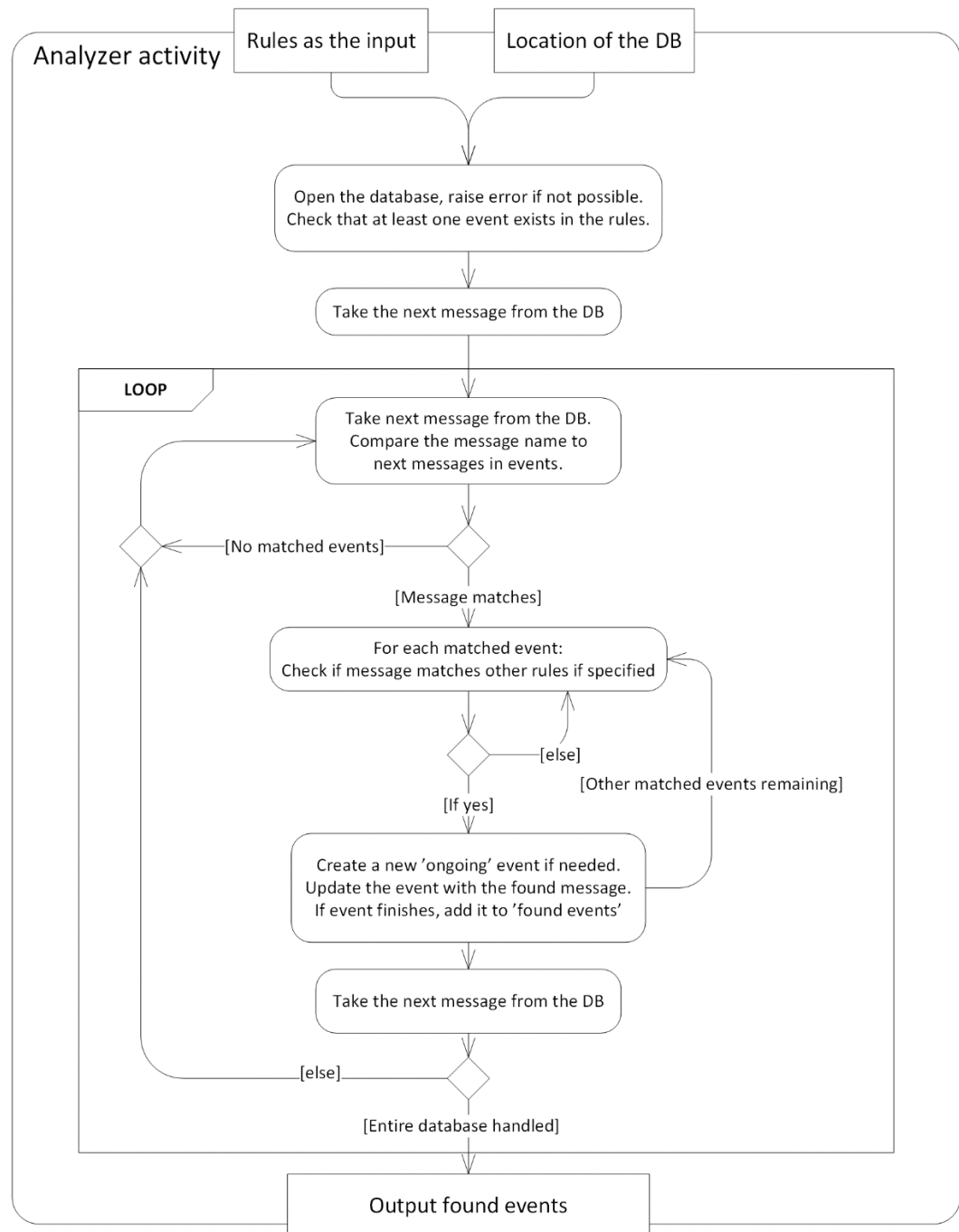


Figure 16: Analyzer internal operation flow

#### 4.6. UI

The UI-module is responsible for displaying the analysis results to the user, including drawing the graphical UI. The UI must fulfil requirements REQ\_2 and REQ\_12. It takes the found events as input. Each found event must include the reaction type (Alert or Visualize) as defined in the rules for the event. The UI handles each found event individually. For 'Alert'-events, the UI shows which messages comprise the event and their indices in the original log along with the event name. 'Visualize'-events could differ in that they are placed on a graphical timeline, where the user can

see where exactly those events begin and end in comparison to other found events, but the version of the GUI designed and implemented in this thesis does not visually differentiate between ‘Visualize’ and ‘Alert’-reactions.

For each event to be displayed for the user, the GUI shows the messages in order of appearance and their timestamp and index. Further message content is not elaborated, even in cases where the event requires certain content within messages. This should keep the GUI easily readable without overloading it with too much information. If the user wishes to check content of a specific message, they can use the index of the message to find it in the log. Each event visually flows from top to bottom, with different events situated side-by-side in the GUI. The GUI supports long events consisting of dozens of messages and high amount of found events with the help sliders in the case that screen real-estate runs out. Original sketch for the layout of the GUI showing found events can be seen in Figure 17, with the arrow symbolizing where the event continues beyond the currently visible messages.


| Event_Name 1   | Event_Name 2                                                                                          | Event_Name 3    | Event_Name 4    |
|----------------|-------------------------------------------------------------------------------------------------------|-----------------|-----------------|
| Msg_1 512 #124 | Msg_6 200 #51                                                                                         | Msg_5 203 #53   | Msg_1 1200 #831 |
| Msg_2 540 #256 | Msg_1 512 #124                                                                                        | Msg_11 720 #512 | Msg_2 1234 #893 |
| Msg_3 562 #285 | Msg_9 563 #290                                                                                        | Msg_5 801 #595  | Msg_3 1251 #901 |
|                | Msg_3 562 #285<br> |                 |                 |

Figure 17: GUI sketch for found events

As the UI module handles each found event individually, each column for the GUI is build one by one. The UI also handles each message of found logs individually. Name, timestamp, and index of the message found to be part of an event is shown as a row within the event column. When all the messages of an individual event have been handled, a new column is created for the next event. The GUI is drawn for the user when all the found events have been handled. The operation flow can be seen in Figure 18.

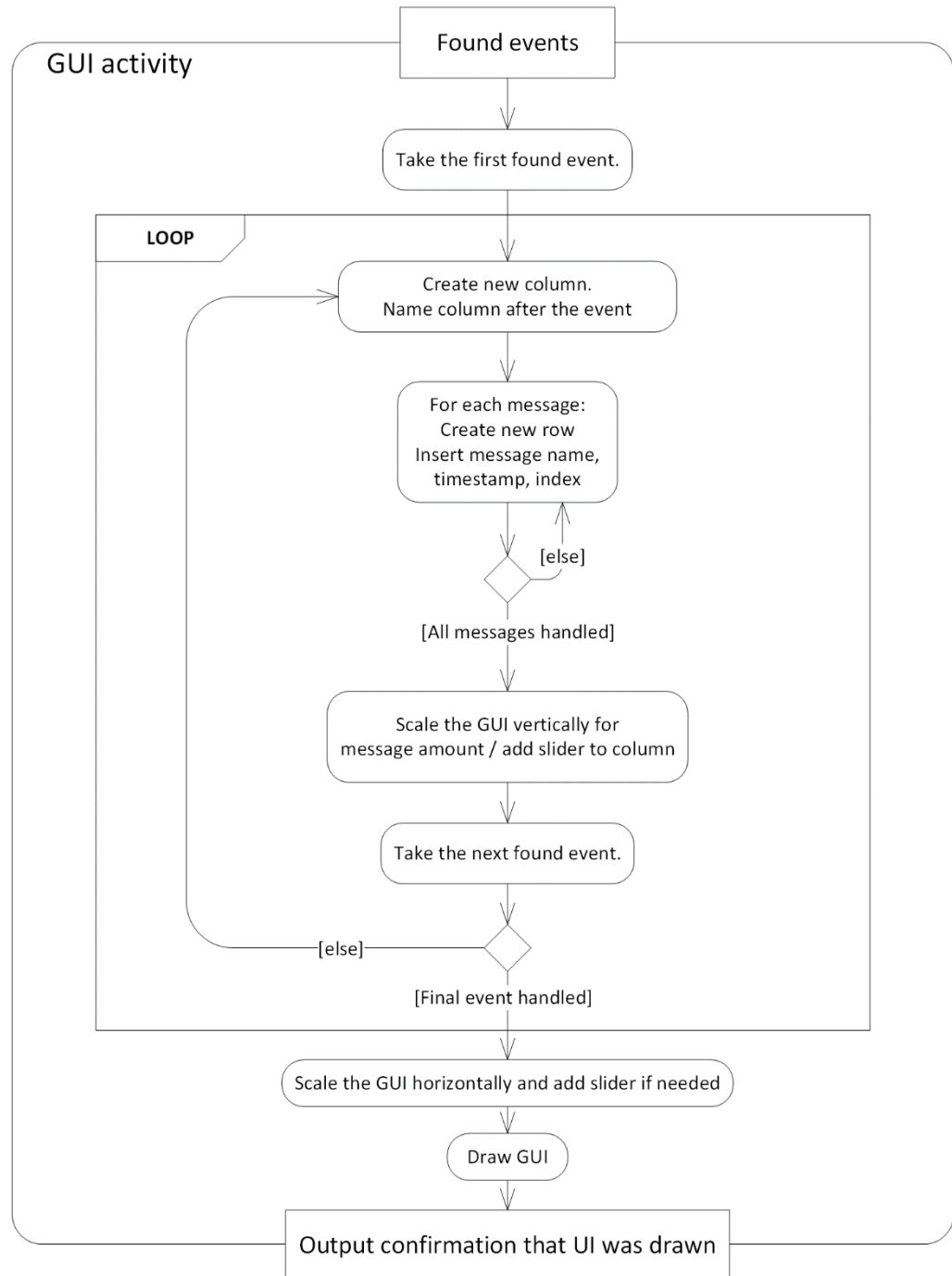


Figure 18: GUI module operation flow

#### 4.7. Summary

This Chapter presented the overarching design of Protocol Log Analyzer. The design is split into four modules, the log parser, the rule reader, the analyzer, and the UI. Protocol Log Analyzer uses rules provided by the user for the analysis. The rules are written using the presented rule notation. For each module, design diagrams and other high-level design principles are presented.

## 5. IMPLEMENTATION

Protocol Log Analyzer is implemented using Python programming language. Python is used as it is flexible, includes a diverse standard library for database and file manipulation, and supports many different platforms and environments. The greatest risk in using Python comes with the performance, as Python programs are usually slower than programs implemented in languages such as C, as tested by Fourment and Gillings [28]. To make sure Protocol Log Analyzer can be run in different devices and environments, only Python's standard library is used. Only utilizing the Python standard library also simplifies the process of modifying it in the future, as the users can be provided with the complete source code for easy modification.

### 5.1. Architecture

The implementation follows the same architecture split presented in the previous Chapter. The log parser, rule reader, and analyzer are implemented in their own Python modules. The GUI is included in the main Python-file. When the main Python-file is run, it calls the other modules as needed. Each module can be run as main on their own, and in those cases simple test that were used during development are run. Users can use those tests as a way to check if their own modifications have the desired effects.

Figure 19 shows the class diagram of the implementation. Error-classes are omitted, as they are not relevant when Protocol Log Analyzer works without issues. The four singleton classes correspond with the modules. The rule reader module also includes *Event*, *Message*, *Delay*, and *PduContent*-classes, which contain the rules extracted from the rule file. *Negation*-class is used to store data about negation-symbols found in the rule file when the rule reader is creating rule events. The analyzer module includes *OngoingEvent*-class, which is used by the analyzer when comparing rule event's to messages found in the log.

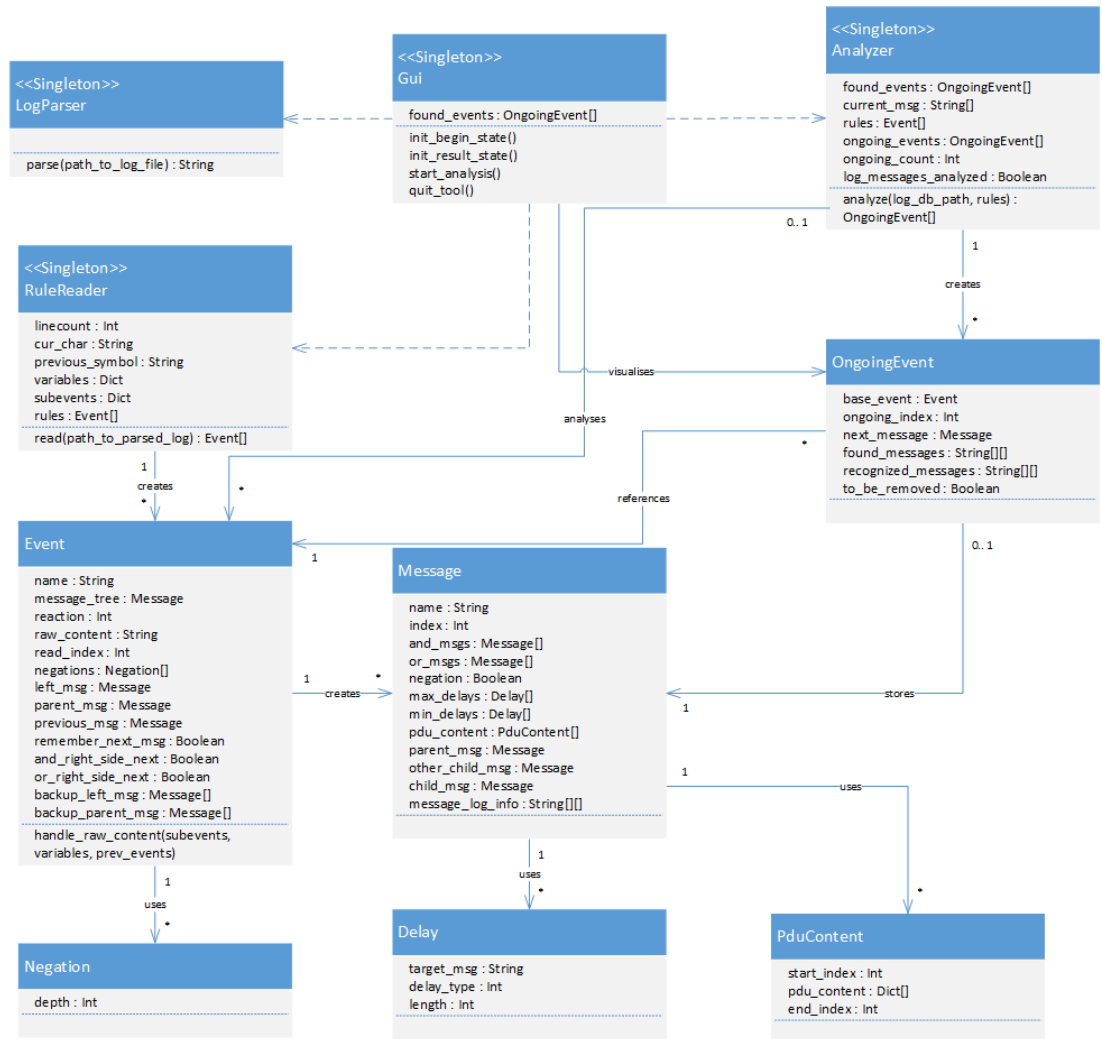


Figure 19: Class diagram of the implementation

## 5.2. Log parser

The log parser is implemented as a class called *LogParser*. When the class is initiated, it creates a folder for the parsed logs. In case that folder already exists, the initiate function does nothing. The relational database chosen for the implementation is SQLite<sup>6</sup>. SQLite works cross-platform and it is popular and stable, which makes it a good option for Protocol Log Analyzer. In Python, SQLite is used with *sqlite3*-module<sup>7</sup>. The database is created when parsing method (*parse*) of *LogParser*-class is called. Table that includes the four fields of the internal representation model is created straight after. If the creation of the table fails, a log with the same name has already been parsed and rest of the parsing sequence is skipped.

For this version of the implementation, the parser is implemented to handle csv-style log files with four fields that match with the fields of the internal representation model. This approach was chosen because the parser would simply ignore any extra fields, thus their presence in the logs is unnecessary, and information learned when making a

<sup>6</sup> <https://www.sqlite.org/index.html>

<sup>7</sup> <https://docs.python.org/3/library/sqlite3.html>

parser for more complex logs might not be applicable to specific proprietary log file formats users would actually use Protocol Log Analyzer with. As described in the design in Chapter 4.2, the parser goes through each message one-by-one, and for csv-logs this means line-by-line. When the whole log has been parsed, the parser returns the path where the database was created. If the parser cannot open the log file, the already created database is deleted and the parser returns an empty path to the database.

### 5.3. Rule reader

The rule reader's implementation is more complicated. It utilizes two classes which have their own methods, its own Error-class called *RuleReaderError*, and four classes that are used to store and manipulate data in their attributes.

First of the classes with methods is *RuleReader*, which provides other modules with a method (*read*) for reading a specified rule file. If the class is provided with a path to a rule file which does not exist, the reading never starts and no rules are found. The reading loop works as described in the design in Chapter 4.4. The rule reader keeps track of how many lines of the rule file have been handled, what rules including variable definitions have already been found, and what was the previous symbol found in the rule file. The rule reading method includes error handling that is triggered when *RuleReaderError* is raised, where the rule reader informs the user how many lines it managed to read before encountering something not deemed valid in the rule file.

The rule reader works with a two-level design. The first level, which corresponds to Figure 14 of the design, looks for comments, variable definitions, and event definitions as those are allowed as top-level operations in rule files. The second level is called when the rule reader encounters an event definition. The second level handles building of event definitions and corresponds to Figure 15 of the design. The second level first creates a new *Event*-object. *Event* is the second class of the rule reader module with methods. It exists as a separate class because the analyzer reuses it. The rule reader reads the entire event definition before handling it, which causes the error handling to not be able to tell the user exactly which line causes an issue in invalid rule file. All whitespaces and comments are removed from the event definitions when the handling starts, which simplifies the rest of the handling procedure. The function for actually handling the event definition is implemented as a method (*handle\_raw\_content*) for the *Event*-object.

The *Event*-class has attributes that it uses to keep track of several variables when building the event from the definition extracted from the rule file and attributes that serve as output for the analyzer. The output attributes are the name of the event, what reaction was defined for it, and a message tree that consists of all possible messages the event can include. Each message in the message tree is an object of *Message*-class. Attributes of message include name of the message, other rules that a message from the log must follow, such as delays and content requirements, and different child messages the message can have in the event. One child message is the main child message, but there exists also another (*other\_child\_msg*) that is used when definition uses brackets, and lists for messages defined with AND and OR-operators. Messages also have an attribute for their parent message, if it exists. These attributes that contain references to other messages allow for travelling up-and-down the message tree.

Rules for specific messages are set immediately after the message is placed in the tree. Messages are set as negated messages if there are an odd number of negations valid for the message when taking brackets into account. When the rule reader

encounters a negation-operator within an event definition, it creates a new negation-object using the *Negation*-class. The attributes of the class tell the rule reader if negations are valid when handling a specific message. This approach requires the rule reader to keep track of where it encountered negations, but it is easy to understand and debug when dealing with several layers of brackets and nested negations. Maxdelay, Mindelay, and Pdu keywords are handled in a loop. The resulting rules are stored in the message object's attributes on lists, which means each message can have an arbitrary amount of extra rules. Classes for both delay and PDU-contents are used to store the rule information within messages in a clear format. Maxdelay and Mindelay keywords use the same class, but are placed to different lists in the message's attributes. If target message is not defined for the delay in the rule file, it is kept empty in the output version as well. How Protocol Log Analyzer should handle those cases is left to the analyzer, because the analyzer has the information on what messages appeared in the log before the current message during runtime. For Pdu keyword the rule reader creates an object which includes the start index, optional end index, and the expected PDU structure, where the string from the rule file is split on each dot. Each substring is then used as a field name when building a nested dictionary, where the innermost field contains either the expected value or Boolean value 'True', depending on if expected value was defined or not.

The rule reader inserts previously defined events or sub-events to later events if it encounters their name as a standalone symbol or word. The current version of the rule reader does not support adding new keywords, such as requiring a minimum or maximum delay for the whole event. This issue could perhaps be sidestepped by inserting the sub-events in a different manner and having the analyzer consider them as single messages instead of a normal part of the message tree, but that would add considerable complexity with little gained utility. Instead, the user can avoid using sub-events in cases where adding new keywords is necessary.

## 5.4. Analyzer

The analyzer module consists of three classes. *Analyzer*-class provides Protocol Log Analyzer with a method for running the analysis and it outputs a list of found events. For input, the analyzer requires path to the log database created by the log parser and list of rules from the rule reader. Found events are objects of *OngoingEvent*-class. Each ongoing event includes a reference to the event rule it matches with, which means the analyzer imports the rule reader module. Final class is error-class used as a catchall for when something unexpected happens during the analysis. Unlike the rule reader, the error handling does not inform the user where Protocol Log Analyzer encountered an issue. This is because errors caused by faulty user input should be caught before the analyzer is run. Errors during analysis instead point towards bugs in the implementation and information about them would be less useful for the end user.

As in the design in Chapter 4.5, the analyzer starts analysis by opening the SQLite database that includes the parsed log messages. The analyzer fetches all items from the database, sorted ascending by their index. Messages are analyzed on a loop until no new messages can be found in the database. At that point, the analyzer returns found events.

When analyzing an individual message, the analyzer starts with checking if the message continues an already existing ongoing event. Check for if the message starts a new ongoing event is done after. Initially the analyzer only adds the information of



the message found in the log to matching messages in events. Updating the ongoing events and creating new ongoing events is done after the message has been marked present.

Log messages are deemed to match with messages in rule events if their names are the same and the rules for delays and PDU-content are not broken. The analyzer checks delay rules by going through previously matched messages on the ongoing event and comparing the analyzed log message's timestamp with the newest message that is targeted by the delay. If the user has not set an explicit target message for the delay with "previous\_msg"-parameter, the analyzer instead tries to compare the timestamps to the preceding message in the rule event. In the implementation this means checking the timestamp of the previous "top-level" message, which can be reached from the start of the message tree by going through main child messages only. Checking all possible messages that the message could target in these cases was deemed not viable, as in certain cases there could be dozens of options. This approach can cause issues when the event rules include other child messages, such as AND and OR-messages. When it comes to defining delays in more complex events, the users can circumvent this issue by defining target messages of the delays explicitly, and the issue should not affect the practical capabilities of Protocol Log Analyzer.

The analyzer checks PDU content by first slicing the PDU string from the log message content with the indices defined in the rule. In the current implementation the string is loaded with Python's standard *json*-module. JSON-format is used in the current implementation due to its easy readability and wide support, but it could be easily replaced in the implementation to support different kinds of log files. This creates a dictionary that the analyzer then compares with the one found in the rule. The comparison supports arbitrary amount of nesting. The comparison checks if each field in the rule exists in the PDU obtained from the log message. If necessary, it compares the specific values of the field found in the final nesting level. If all the fields required in the rule are found in the PDU and values satisfy the requirements, the PDU content rule is fulfilled.

The analyzer creates new ongoing events and updates status of already existing ongoing events after log message information has been updated. Each ongoing event keeps a track of what log messages have been verified to be part of it, which log messages match with its rules, and to what point in the rule event has the log already fulfilled the event rules. From the analyzer point-of-view, the message tree of the event rule consists of levels. Each level includes the main message and its child messages except for the main child message. The main child message is the main message of the next level of the message tree. Each level can include an arbitrary amount of messages but in practice, it would be simpler for the user to define several different event rules instead of creating a single complex one. Level is considered detected from the log if all necessary messages have log message information attached, or in the case of negation messages, not attached. If the level includes child messages, the analyzer takes them into consideration while assessing if the level has been detected or not. If the analyzer considers the level detected, the next level of the message tree is set as the messages the analyzer compares following log messages with, and list of log messages that have been verified to be part of the event is updated to correspond with the log messages that the analyzer thinks fulfilled the levels requirements. An exception to this comes with negation messages as if they would be assessed with the same logic, the analyzer would skip those levels immediately. Instead, negation message levels are considered detected only after the level following them has been

deemed detected. This level logic for assessing the events allows the analyzer to keep track of multiple different options for messages that would fulfill the rule event while making it possible to easily discard not fulfilled options such as OR-message routes that were not found in the log.

The analyzer creates a new ongoing event if enough log message information has been added for the rule event's first level to be considered detected in the log. The rule event is deep copied to the new ongoing event and the first level is marked as already detected. When analyzing subsequent log messages, ongoing messages are thus always at least on second level of the message tree. After the creation of the new ongoing event has finished, the analyzer removes log message information from the rule event. This is because that information is now stored in the ongoing event and their presence in the rule event would lead to the analyzer creating a new ongoing event for every subsequent analyzed log message regardless of their content.

Ongoing events are removed ongoing events list if they are completed or are judged to be impossible to complete. Events are completed when the final level of the message tree has been detected from the log. Events are impossible to complete when they include defined maximum delays and the timescales in those delays have already expired when comparing to the timestamp of the latest log message analyzed, or if messages which event requires not to exist have been detected in the log. Removing 'useless' ongoing events is done because they would otherwise clutter the analysis process, hurting performance and leading to longer processing times. Current implementation still leads to a large amount of ongoing events existing during the analysis, but it is not expected to lead to meaningful performance issues.

## 5.5. Main and UI

The main module of the implementation includes a class for the graphical user interface. The UI module does not exist as a separate module as that would essentially leave no purpose for the main module. The UI has two different views, with first view used to choose the analyzed log and the rule file used for analysis, and the second view displaying the user with the results.

The UI is implemented with *tkinter*-package<sup>8</sup>, which provides a Tcl/Tk GUI toolkit interface for Python. Unlike other GUI toolkits such as GTK, Tkinter is part of a standard Python installation. *Gui*-class includes methods for building the UI and utilizing all other modules of the implementation. When the user starts Protocol Log Analyzer, the UI is loaded with the start view visible. The start view can be seen in Figure 20. This view includes four buttons. One button is for choosing a log file and another for choosing a rule file, which open an OS native file manager window. Two lower buttons are for starting the analysis and closing the program. When the user has chosen the files, they can start the analysis process whenever they choose. When the user starts the analysis process, Protocol Log Analyzer first runs the parser, then reads the rule file. If it finds no rules in the rule file, the analyzer is not run, as it would be useless. Instead, it shows an error about the lack of rules to the user. If rules are found, the analysis proper is performed and the UI moves to result view.

---

<sup>8</sup> <https://docs.python.org/3/library/tkinter.html>

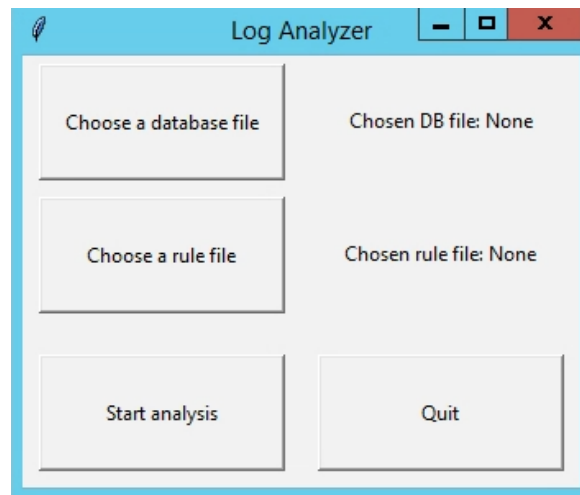


Figure 20: Start view of the UI

The design in 4.6 describes the result view. Columns and rows for each event are created with *Listbox*-widgets. Each column is its own listbox. Listboxes include vertical “sliders” by default. Vertically the UI limits the listboxes to 20 lines and first two lines are reserved for the name of the found event and an empty line that makes the UI easier to read. This means that the UI can show found events as long as 18 messages before the user has to scroll down further on the listbox. Support for showing more than a few events is done by adding a horizontal slider to the listboxes. In addition to the information about the found events, the result view includes a button for returning to the start view and a button for closing Protocol Log Analyzer.

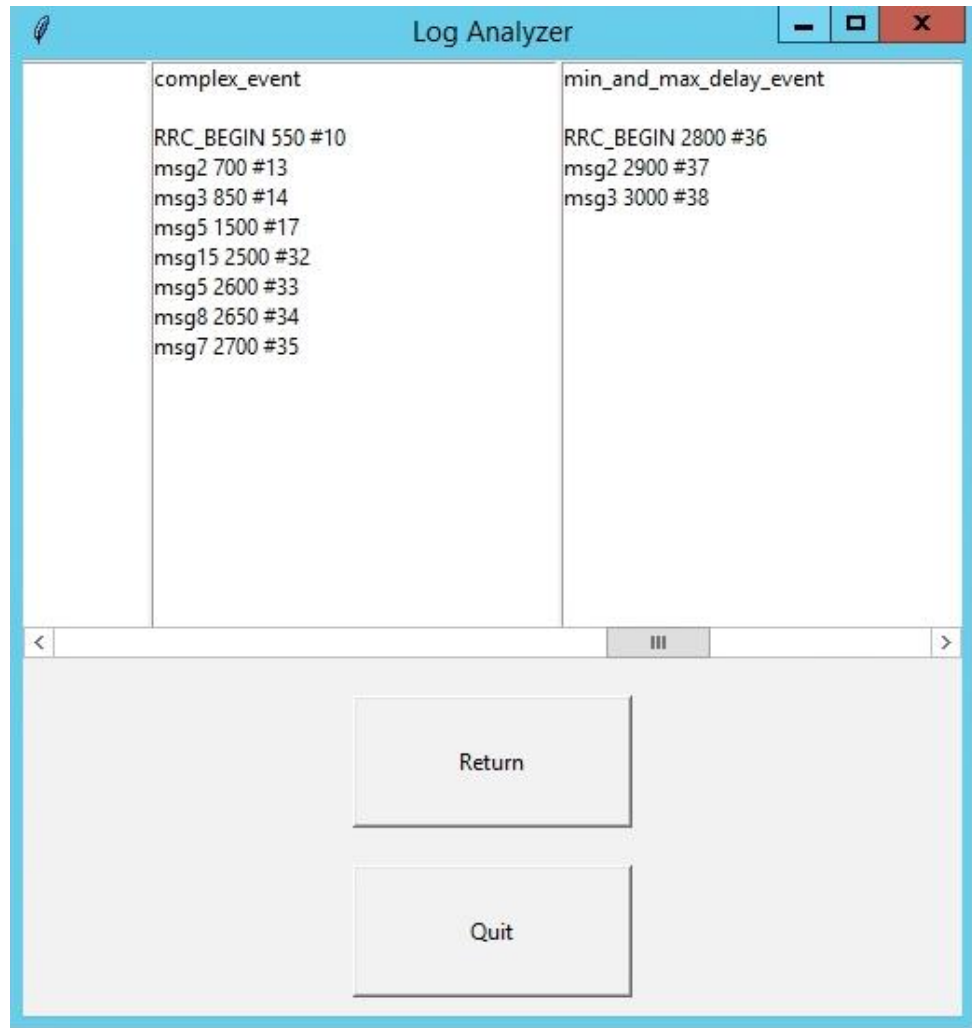


Figure 21: Result view of the UI with some testing data

## 5.6. Summary

This Chapter presents an implementation based on designs of Chapter 4. Log parser, rule reader, analyzer, and UI modules are presented, along with the interfaces they provide to other modules. These modules are distributed as default Python source code and running them requires a complete Python installation. Development was done using Python version 3.4.3. Packaging Protocol Log Analyzer as a singular executable would also be possible.

## 6. RESULTS

This Chapter describes the evaluation of overall results for the implementation. Protocol Log Analyzer was evaluated from the perspective of a user who writes their own rule set, meaning the process of writing the rules and analyzing a log with Protocol Log Analyzer are performed. The goal of the evaluation was to demonstrate the utility of the developed rule notation, locate issues present in the current design of rule notation and Protocol Log Analyzer, and test for performance issues at least with relatively small logs.

### 6.1. Evaluation preparation

Some type of log is required to test the functionality of Protocol Log Analyzer. For this evaluation, the log was created artificially. Artificial log was used because it could be created in a format that could readily be parsed by the log parser, and it allowed for more freedom regarding the content. First, a Python script for creating a csv-style log file was created. The script writes 1000 rows with four fields that match with the fields of the internal representation model. The length of 1000 log messages was chosen to represent a modem log that has already underwent preprocessing filtering out unwanted log messages. Index for the log messages increases by one for each new message, timestamp increases by 10, and name of the message is string “*msg*” with the index added. Content of each message is left empty. These programmatically created log messages make up for most of the artificial log. Their purpose is to verify that the Protocol Log Analyzer handles large swathes of data that is irrelevant to the analysis results well.

Certain programmatically created log messages were replaced with fake RRC messages and updates regarding UE’s state. The fake messages were inspired by NR RRC message sequences in TS 38.508-1 [29] and TS 38.523-1 [12] specifications. The log message names of fake RRC messages correspond with the names of real RRC PDUs, but content within the PDUs is not the same. In certain cases, PDU-content is not present at all and in other cases only partial PDU-content is added to the log messages. This is done because the advantage from having a complete PDU is miniscule when compared to the needed workload to create valid PDUs considering that Protocol Log Analyzer only reads the fields specified in the rules. The RRC procedures chosen for the fake RRC messages were connection establishment, access stratum security activation, network initiated connection release, UE capability transfer, DRB1 configuration, and handover from one NR cell to another. The procedures were inserted to the artificial log in realistic order, for example security activation follows connection establishment. Other manually inserted log messages were log messages which signal the UE changing its state. These were inserted in locations where RRC procedures would cause the UE to change its state. Content of those messages is always empty.

### 6.2. Creating the rule set

Creating the rule set works both as a preparation for evaluation of how Protocol Log Analyzer runs and as evaluation for the rule notation. Writing rules for NR RRC procedures described in the 3GPP specifications [11] [29] [12] should reveal possible

flaws and deficiencies in the rule notation. Same procedures that were inserted to the artificial log were used as basis for creating the rules. It is important to note that knowledge of what type of data is available in the log is necessary for writing the rules, thus writing the rules just based on the specifications is not possible. In this case it means taking into account what PDU-content is present in the fake RRC messages in the artificial log and focusing on rules that utilize that content.

First rules relate to UE's states. The user would like to know at what times the UE changes state. In the artificial log, there are special log messages which indicate UE state changes. The rules are written so that the user always knows at which state the UE is at any point in the log. In Figure 22, a rule for recognizing when UE enters RRC\_CONNECTED state is presented. As the entire events consists of a single log message, the event is recognized whenever the log message signifying UE's state changing to RRC\_CONNECTED is encountered. When two similar events are created for the other two states, the user should be able to see from the analysis results what the most recent state change event is and thus, what is the state of the UE at any given point.

```
/* UE enters RRC_CONNECTED state */
Event enter_connected =
{
    Message("RRC_ENTER_CONNECTED")
};
```

Figure 22: Event where UE's state changes to RRC\_CONNECTED

When it comes to RRC procedures, these state change messages can be used to make sure the UE is in the correct state before and after the procedures. A rule for recognizing successful initial access stratum security activation, a procedure described in Chapter 5.3.4 of TS 38.331 [11], requires the UE to be in RRC\_CONNECTED state when it receives *SecurityModeCommand*-message from the network. The rule can be seen in Figure 23. The rule defines the correct state with the help of negation-messages for IDLE and INACTIVE states. Appearance of those messages would mean the UE has changed state away from RRC\_CONNECTED, which leads to the message sequence in the log not matching with the procedure. The RRC messages expected for the rule are defined after. For RRC connection release, it is important that the state of the UE matches the expected both before and after *RRCRelease*-message. This can be also be seen in Figure 23, where a rule based on Chapter 8.1.1.3.1 of TS 38.523-1 [12] is defined, including a 60 millisecond delay after receiving *RRCRelease*.

```

/* Access stratum security activation */
Event as_security_activation_success =
{
    Message("RRC_ENTER_CONNECTED"),
    !Message("RRC_ENTER_INACTIVE") & !Message("RRC_ENTER_IDLE"),
    Message("SecurityModeCommand"),
    Message("SecurityModeComplete")
};

/* RRC connection release */
Event release =
{
    Message("RRC_ENTER_CONNECTED"),
    !Message("RRC_ENTER_INACTIVE") & !Message("RRC_ENTER_IDLE"),
    Message("RRCRelease"),
    Message("RRC_ENTER_IDLE", Mindelay(60))
};

```

Figure 23: Events for security activation and RRC release

Checking of PDU-content is used in certain rules. When defining rules for UE capability transfer adapted from Chapter 8.1.5.1.1 of TS 38.523-1 [12], the rule requires *UECapabilityEnquiry* and *UECapabilityInformation*-messages to have their *rat-Type*-fields set to value “0”, which corresponds to “nr”-value of the specification. If those fields are not present in the PDU, or their values are not as specified, the rule should not be fulfilled when analyzing. The rule cannot match with the specification exactly, as the rule notation does not support defining content that is not allowed to be present in the PDU, like the specification describes for contents of *UECapabilityEnquiry*, and it also does not support checking that transaction identifiers of between two different messages are equal.

Rules can use PDU-content to differentiate between events that share log message names. This is used for *RRCReconfiguration*-messages of DRB1 configuration and intra NR handover, with rules presented in Figure 24. In DRB1 configuration described in Chapter 8.1.2.1.1 of TS 38.523-1 [12], the *RRCReconfiguration* includes exactly one *RLC-Bearer-Config* IE in *rlc-BearerToAddModList* field of *masterCellGroup* field. The notation does not support defining length of arrays found in PDUs, so this part of the specification is ignored. *RRCReconfiguration* must also include a *RadioBearerConfig* IE in *radioBearerConfig* field. In comparison, in intra NR handover, presented in Chapter 8.1.4.1.2 of TS 38.523-1 [12], a *RRCReconfiguration* contains *measObjectToRemoveList* in *measConfig* with two *measObjectIds* in a specific order with different expected values in the list. The closest approximation rule notation supports is defining the expected value twice, but defining the order is not possible. Expected contents of *masterCellGroup* fields differ significantly. In intra NR handover it contains *spCellConfig*, which is a field that shall not be present in DRB1 configuration. This difference however cannot be written in the rules with the current version of the notation, as it is limited to only defining what values and fields must be present. Outside of the contents of *RRCReconfiguration*, the two created rules are

identical, with requiring the UE to be in `RRC_CONNECTED` state beforehand and having *RRCReconfigurationComplete* follow after *RRCReconfiguration*.

```
/* DRB1 configuration */
Event drb1_reconfiguration =
{
    Message("RRC_ENTER_CONNECTED"),
    !Message("RRC_ENTER_INACTIVE") & !Message("RRC_ENTER_IDLE"),
    Message("RRCReconfiguration", Pdu(0, criticalExtensions.rrcReconfiguration.
        nonCriticalExtension.masterCellGroup.
        rlc-BearerToAddModList.servedRadioBearer.drb-Identity == 1),
        Pdu(0, criticalExtensions.rrcReconfiguration.radioBearerConfig)),
    Message("RRCReconfigurationComplete")
};

/* Intra NR handover reconfiguration */
Event intra_ho =
{
    Message("RRC_ENTER_CONNECTED"),
    !Message("RRC_ENTER_INACTIVE") & !Message("RRC_ENTER_IDLE"),
    Message("RRCReconfiguration", Pdu(0, criticalExtensions.rrcReconfiguration.
        nonCriticalExtension.measConfig.measObjectToRemoveList == 1),
        Pdu(0, criticalExtensions.rrcReconfiguration.nonCriticalExtension.
        measConfig.measObjectToRemoveList == 2),
        Pdu(0, criticalExtensions.rrcReconfiguration.nonCriticalExtension.
        measConfig.measIdToRemoveList == 1),
        Pdu(0, criticalExtensions.rrcReconfiguration.nonCriticalExtension.
        masterCellGroup.spCellConfig.reconfigurationWithSync.
        rach-ConfigDedicated.Uplink),
        Pdu(0, criticalExtensions.rrcReconfiguration.nonCriticalExtension.
        masterKeyUpdate.keySetChangeIndicator == 1),
        Pdu(0, criticalExtensions.rrcReconfiguration.nonCriticalExtension.
        masterKeyUpdate.nextHopChainingCount == 0),
        Pdu(0, criticalExtensions.rrcReconfiguration.nonCriticalExtension.
        masterKeyUpdate.nas-Container)),
    Message("RRCReconfigurationComplete")
};
```

Figure 24: RRC reconfiguration event rules

In addition to rules described above, certain very general rules were also created, such as a generic reconfiguration without any PDU-content limitations. In total, the rule file includes 97 lines of notation for 12 different rule events. Creating the rules exposed some defects in the rule notation regarding how the user can define the expected contents. Defining the overall structure of events according to RRC procedures is however quite simple and the resulting rules are easy enough to read and modify. When writing rules for similar events with differing keywords, the user has to repeat a lot of definition as sub-events cannot be used in cases where keywords would have to be inserted within the sub-event.



### 6.3. Analyzing the artificial log

Once the rules had been written, Protocol Log Analyzer was used to analyze the created artificial log. Expectation for this was that Protocol Log Analyzer finds the state changes and log message sequences corresponding to RRC procedures. The time Protocol Log Analyzer spends on parsing the log file, reading the rule set, and performing the analysis was measured, with the rule set being tweaked in order to see if its complexity affects the overall performance. Measurements were done with Python *time*-module<sup>9</sup>, part of the standard library. For each variant of the rule set, the measurements were performed 5 times (N=5). The database created from the parsed log was deleted after each measurement to make sure the parser parses the log separately each time. The purpose of the measurements was to get an initial view on which modules require the most execution time, catch possible major performance issues, and see how the amount of rules affects the time needed for analysis. The figures should not be taken as estimates for the execution time of Protocol Log Analyzer in other environments or with a different rule or log file. The results for these measurements can be seen in Table 8. The execution time of analyzer is the most interesting, as it should vary with the complexity of the rule file.

Protocol Log Analyzer with all 12 rules set with “Alert”-notification found 28 events in the artificial log. These 28 events were all the expected events, though some RRC procedures present in the log are recognized several times, as there is overlap with the rule events, for example the same message sequence can be detected as both Intra NR handover and generic reconfiguration. Going through all 28 events with the GUI proved to be cumbersome.

For the second measurements, the rule events notifying the user about UE state changes were modified by setting their notification to “None”. This makes them invisible to the analyzer, though they are still handled by the rule reader. In total, this meant there were 9 rules given to the analyzer and it found 16 events.

For the third and final measurements, notification were set to “None” for certain rules depicting RRC procedures in addition to UE state rules. This brought the total amount of rules given to the analyzer down to 6. With these rules, Protocol Log Analyzer found 11 events in the log.

Table 8. Module execution time with different variants of the rule file

| Amount of rule events | Total time (mean ms) | Parser (mean ms) | Rule reader (mean ms) | Analyzer (mean ms) |
|-----------------------|----------------------|------------------|-----------------------|--------------------|
| 12                    | 227, $\sigma=22$     | 94, $\sigma=20$  | 13, $\sigma=3$        | 115, $\sigma=7$    |
| 9                     | 215, $\sigma=12$     | 91, $\sigma=7$   | 15, $\sigma=3$        | 104, $\sigma=6$    |
| 6                     | 189, $\sigma=11$     | 83, $\sigma=6$   | 13, $\sigma=3$        | 88, $\sigma=4$     |

Performance measurements did not reveal any major performance issues. With a relatively small log and rule file, the total execution time was around 200 milliseconds. When it comes to user experience, that time is very short, as the information Protocol Log Analyzer provides its user could significantly decrease the time it takes for the user to finish the analysis manually. The length of execution time for each module is visualized in Figure 25. It is clear that in the tested conditions parsing and analysis take the majority of overall execution time.

<sup>9</sup> <https://docs.python.org/3/library/time.html>

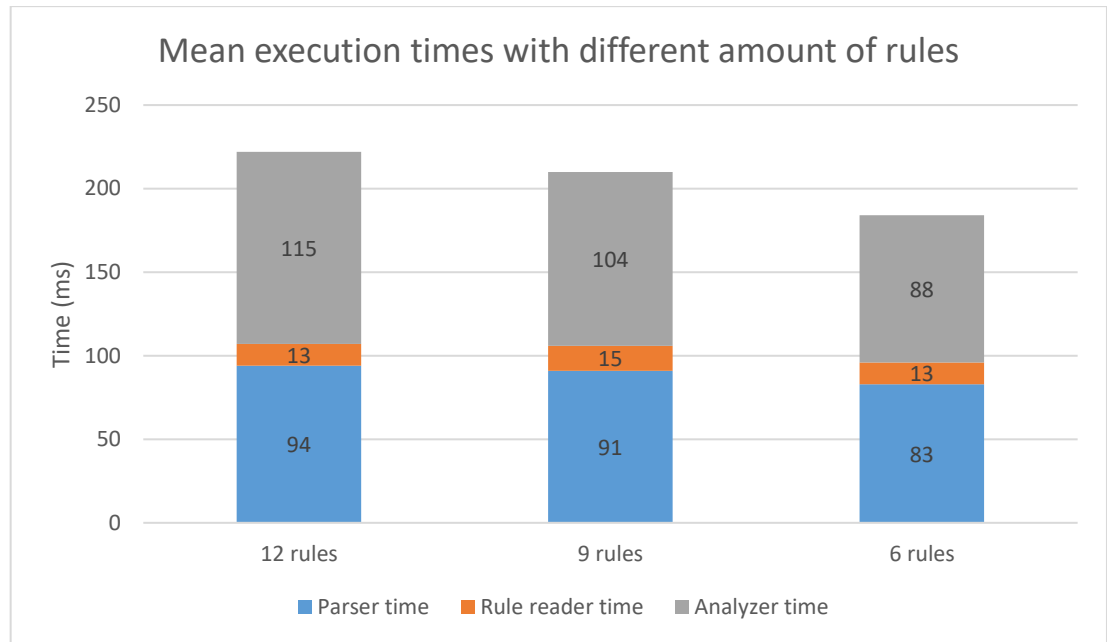


Figure 25: Share of execution time of different modules

When the amount of rule events provided for the analyzer is reduced, its execution time also reduces. This is visualized in Figure 26, where the measured analyzer execution times are marked with the mean line showing the trend of execution time being longer with more rule events. From the visualization, the spread of measured analyzer execution times can be seen. There are no significant differences when it comes to spread of execution times for different amount of rule events. Reduction of execution time with less rule events is most likely because the analyzer then has to start significantly less ongoing events, and compare their next expected message to following handled log messages. As the meager amount of 12 rules has significant impact on the overall execution time, the amount of rules could be bigger factor in overall execution time than the length of the log. Overall, Protocol Log Analyzer performed according to expectations, and similar performance in real life use cases would mean Protocol Log Analyzer is helpful for the users.

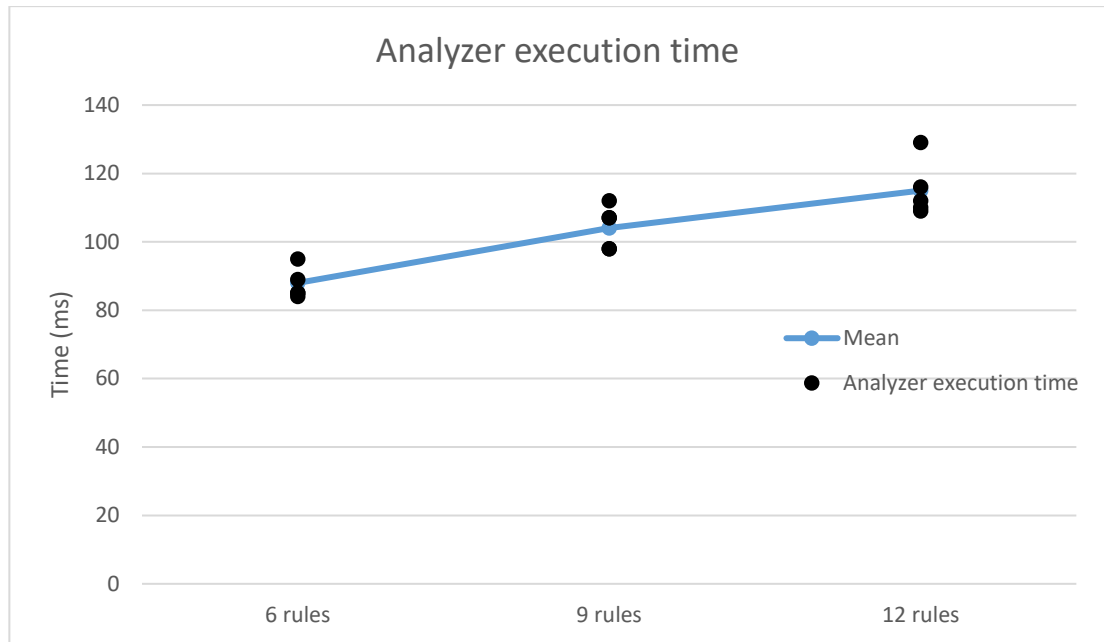


Figure 26: Analyzer execution time on different runs

#### 6.4. Issues and possible solutions

Though Protocol Log Analyzer and the rule notation perform well overall, some issues arose during the evaluation. Issues concerning the rule notation affect also the Protocol Log Analyzer itself.

The lack of support for defining variables during analysis is a significant oversight. Without it, the rule notation cannot be used to check that transaction identifiers match between request and responses. Adding this feature might not even require a new keyword, as just using a previously undefined variable during an Event-definition could be defined to mean that the variable is filled from the log during runtime.

Another rule notation issue is that it does not support defining content that is not allowed to be present in PDUs. Simple fix at the rule notation level would be to add support for “NOT”-operator (!) with Pdu-keyword. This would allow the user to define mutually exclusive rule events easier, as field that must be present in message of Event A, would not be allowed at all in message of Event B.

Another lacking feature when compared to 3GPP standards is the inability to define lists. Current version of the rule notation and Protocol Log Analyzer handle lists by ignoring their structure and comparing all possible indices of PDU-content to expected values. This works fine in most cases, but some standard definitions expect the lists to be in certain order and of certain length. New keyword for list length and support for defining expected indices is needed in the rule notation.

When it comes to performance, the biggest question is would a large amount of ongoing rules lead to execution time imploding. This could happen due with rules that have the first level of their message tree very often recognized in the log, but are rarely invalidated or completed by the following log messages. If this becomes a performance issue, the analyzer could be reworked to only start ongoing events after several levels of the message tree have been recognized, hopefully cutting down on the amount of ongoing events, or by setting a hard limit on how many ongoing events of the same

type can exist at the same time. Both approaches would have a negative impact on the accuracy of Protocol Log Analyzer.

GUI being cumbersome when there are a lot of found events suggest that implementing support for “Visualize” if of high priority. As it is already included in the rule notation and taken into account in the design, changes are only needed in the GUI. Other option would be to add a priority setting for the rule events and only show for example 10 found events with the highest priority. The priority setting could also be used to make sure the GUI only shows the most precise version of a rule event in cases where a presence of a more generic rule is implied when another event is found. The rule events could be linked to more or less precise variants of the same event at the rule notation level with some kind of “child event”-option.

The current implementation of sub-events cannot be used if any changes are needed within the messages of the sub-event. This does not affect the capabilities of rule notation or Protocol Log Analyzer regarding the analysis, but it can force the user to repeat a lot when writing the rules. Sub-events could be updated to take “arguments”, which could then be inserted to positions defined in the sub-event definition. This would however require significant changes to both rule notation and Protocol Log Analyzer implementation, as those sub-events would not be valid standalone and they would need to be inserted to other events by the rule reader in a very different fashion compared to the current implementation.

## 7. DISCUSSION

Protocol Log Analyzer presented in this thesis is very much a proof-of-concept. It does not work with any real logs, but the methods chosen show promise. The rule notation can be used to define wanted and unwanted message sequences Protocol Log Analyzer looks for in the analyzed log in a way that is familiar for users due to similarities to C-language [27] and EBNF [1]. As Jayathilake [14] mentions, lack of standards regarding log files constitutes an issue, and Protocol Log Analyzer assumes that its parser is modified when necessary to provide accurate parsing for different types of logs. RRC messages and PDUs presented in TS 38.331 [11] are used as examples of what kind of data could be found in cellular modem logs. The rule notation can be used, with some exceptions, to define expected PDU-content according to 3GPP RRC standards, and the developed analyzer can compare expected PDU-content to what is found in the analyzed log. Overall, Protocol Log Analyzer and rule notation differ from previously presented log analyzers, such as Swatch [19], by giving the users the ability to define message sequences to be looked for in the log. When given the ability to programmatically look for sequences from the logs, the users can cut down the amount of effort required when analyzing logs.

Development and evaluation was limited by the fact that logs tend to be proprietary and when it comes to 5G cellular modem logs, there are not many different types available. Protocol Log Analyzer is not actually limited to be used with cellular modem logs, but development based on another type of log could have led to the design to overlook traits of 3GPP PDUs. Focusing on areas that are transferable between different logs, such as the rule notation and the analyzer, allows for easier future development. Nonetheless, the lack of results from using Protocol Log Analyzer on a real log limits the confidence of any statement about the utility of Protocol Log Analyzer.

Lessons learned include focusing on what distinguishes the developed log analyzer earlier on. When the decision was made to focus on the rule notation and analysis methods instead of parsing, a lot of background research into different types of existing parsers was already done. Most of that research had very little direct impact on the final developed Protocol Log Analyzer. Instead, interfaces for the different modules could have been defined earlier and the focus could have been on the interworking of the analyzer and rule notation. This would include evaluation that is more extensive and would thus provide more concrete results.

Next phase for the development of Protocol Log Analyzer is to get it to work with a proper cellular modem log. This requires significant changes in the log parser, but it would allow for Protocol Log Analyzer to be evaluated in a real use scenario. User feedback would then be important in tweaking the rule notation and Protocol Log Analyzer to improve on how users define rules and analyze logs accurately and efficiently. This would include developing the GUI for better user experience to better fulfill the goal of Protocol Log Analyzer being useful for daily activities of the users. The rule notation should be presented to users to see what kind of rules they write with it and if that exposes flaws or missing features.

Long-term future work would be to try to integrate automated log analysis techniques to writing the rules, log parsing, or log analysis. Automated rule writing could ease the workload of the users by generating either partial or complete rules based on previous logs. Automated techniques in the parser could allow Protocol Log Analyzer to be adapted to work with different types of logs more easily, assuming the

parser could locate the name of the message, timestamp, index, and content from the log file automatically.

## 8. CONCLUSION

Devices such as smartphones use cellular modems for wireless communication. Cellular modems use telecommunications protocols, which define rules for wireless communication methods. Sometimes, cellular modems fail and engineers are required to diagnose issues. To help with the engineers' task, cellular modems write log files, which contain information about the events of the modem. These modem logs can be very large and the analysis process time consuming, so methods to automate parts of the process are needed. Tools for log parsing and log analysis are not a new thing, but previous implementations lack some features required in the modem log analysis.

In this thesis, design and development of a cellular modem log analysis tool called Protocol Log Analyzer and a metasyntax notation for rule definition are presented. Requirements for Protocol Log Analyzer and the rule notation are based on wants of cellular modem engineers. Traits of 3GPP telecommunication protocols are presented and then taken into account during the design process. Radio Resource Control protocol of 5G New Radio is used as an example of a telecommunications protocol. Both Protocol Log Analyzer and the rule notation support expected message sequences, expected PDU-content, and maximum and minimum delays between log messages.

Focus on the development of Protocol Log Analyzer and the rule notation is on performing the analysis and giving users tools to define analysis rules. Design is split into four different modules, the log parser, the analyzer, the rule reader and the rule notation, and the UI. Operation flow for each module is presented. The rule notation is inspired by already existing notations, but is focused on features needed in modem log analysis. Complete keyword and operator list for the rule notation is presented. The implementation follows the same module split as the design. The implementation presentation includes class structure of each module as it is in the developed Python-modules.

Protocol Log Analyzer is not usable with real modem logs. All development and testing was done with artificial modem logs, that include messages as defined in RRC protocol standards. Adapting Protocol Log Analyzer to work with real modem logs would require changes to the log parser at least. During evaluation, Protocol Log Analyzer is used to analyze an artificial log of 1000 log messages. The artificial log includes log messages corresponding to RRC protocol messages and other modem events. Evaluation revealed some lacking features in the current version of the rule notation regarding defining lists and unwanted fields within PDUs. Performance of Protocol Log Analyzer is good within the evaluation environment. Methods for defining telecommunications protocol rules for log analysis, such as message sequences and PDU-content, work as intended and flaws detected in the rule notation are fixable. If Protocol Log Analyzer and the rule notation perform similarly in real life use scenarios, it would be a great help for the engineers. The next step in development of these methods is updating Protocol Log Analyzer to work with real modem logs and seeing how it and the rule notation perform in daily activities of cellular modem engineers.

## 9. REFERENCES

- [1] International Organization for Standardization, "ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF," 1996.
- [2] R. Sharp, Principles of protocol design, Springer, 2008.
- [3] D. Hercog, Communication Protocols: Principles, Methods and Specifications, Springer Nature, 2020.
- [4] G. J. Holzmann and W. S. Lieberman, Design and validation of computer protocols, Prentice hall Englewood Cliffs, 1991.
- [5] O. Dubuisson, ASN. 1 communication between heterogeneous systems, Morgan Kaufmann, 2000.
- [6] A. Toskala, "Background and Standardization of WCDMA," in *WCDMA for UMTS: Radio access for third generation mobile communications*, John Wiley & Sons, 2005, pp. 61-74.
- [7] H. Holma, A. Toskala, T. Nakamura and T. Uitto, "Introduction," in *5G technology: 3GPP new radio*, John Wiley & Sons, 2020, pp. 1-11.
- [8] A. Toskala and M. Poikselkä, "5G Architecture," in *5G technology: 3GPP new radio*, John Wiley & Sons, 2020, pp. 67-86.
- [9] 3GPP Technical Specification TS 38.300. NR; NR and NG-RAN Overall description; Stage-2, Version 16.4.0, 2021.
- [10] T. Henttonen, J. Koskela, B. Sébire and A. Toskala, "5G Radio Protocols," in *5G technology: 3GPP new radio*, John Wiley & Sons, 2020, pp. 149-186.
- [11] 3GPP Technical Specification TS 38.331. NR; Radio Resource Control (RRC); Protocol specification, Version 16.3.1, 2021.
- [12] 3GPP Technical Specification TS 38.523-1. 5GS; User Equipment (UE) conformance specification; Part 1: Protocol, Version 16.6.0, 2021.
- [13] P. He, J. Zhu, S. He, J. Li and M. R. Lyu, "An Evaluation Study on Log Parsing and Its Use in Log Mining," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.
- [14] D. Jayathilake, "Towards structured log analysis," in *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*, 2012.
- [15] K. Kc and X. Gu, "ELT: Efficient Log-based Troubleshooting System for Cloud Computing Infrastructures," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, 2011.
- [16] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014.
- [17] L. Mariani and F. Pastore, "Automated Identification of Failure Causes in System Logs," in *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, 2008.



- [18] P. He, J. Zhu, Z. Zheng and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017.
- [19] S. E. Hansen and E. T. Atkins, "Automated System Monitoring and Notification with Swatch," in *Proceedings of the 7th USENIX Conference on System Administration*, 1993.
- [20] D. Jayathilake, "A mind map based framework for automated software log file analysis," in *International Conference on Software and Computer Applications*, 2011.
- [21] L. Mariani and M. Pezze, "Dynamic Detection of COTS Component Incompatibility," *IEEE Software*, vol. 24, pp. 76-85, 2007.
- [22] A. Sellink and C. Verhoef, "Development, assessment, and reengineering of language descriptions," in *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, 2000.
- [23] J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," in *Proceedings of the International Conference on Information Processing*, 1959.
- [24] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois and others, "Revised report on the algorithmic language Algol 60," *The Computer Journal*, vol. 5, pp. 349-367, 1963.
- [25] D. E. Knuth, "Backus Normal Form vs. Backus Naur Form," *Commun. ACM*, vol. 7, no. 12, pp. 735-736, 1964.
- [26] E. F. Codd, "Relational database: A practical foundation for productivity," *Commun. ACM*, vol. 25, no. 2, pp. 109-117, 1982.
- [27] International Organization for Standardization, "Expressions," in *ISO/IEC 9899:1999 Programming languages — C*, 1999, pp. 67-94.
- [28] M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics," *BMC Bioinformatics*, vol. 9, no. 1, p. 82, 2008.
- [29] 3GPP Technical Specification TS 38.508-1. 5GS; User Equipment (UE) conformance specification; Part 1: Common test environment, Version 16.6.0, 2020.