Lappeenranta University of Technology
Faculty of Technology Management
Department of Information Technology

Master's Thesis

Antti Pohjonen

TEST AUTOMATION SCHEME FOR LTE CORE NETWORK ELEMENT

Examiners of the Thesis:	Professor Jari Porras M.Sc. Henri Elemo
Instructor of the Thesis:	M.Sc. Henri Elemo

ABSTRACT

Lappeenranta University of Technology
Department of Information Technology

Antti Pohjonen

TEST AUTOMATION SCHEME FOR LTE CORE NETWORK ELEMENT

Thesis for the Degree of Master of Science in Technology

2010

92 pages, 27 figures and 3 tables

Examiners: Professor Jari Porras
M. Sc. Henri Elemo

Keywords: Test automation, LTE, Software testing, Agile development

Modern sophisticated telecommunication devices require even more and more comprehensive testing to ensure quality. The test case amount to ensure well enough coverage of testing has increased rapidly and this increased demand cannot be fulfilled anymore only by using manual testing. Also new agile development models require execution of all test cases with every iteration. This has lead manufactures to use test automation more than ever to achieve adequate testing coverage and quality.

This thesis is separated into three parts. Evolution of cellular networks is presented at the beginning of the first part. Also software testing, test automation and the influence of development model for testing are examined in the first part. The second part describes a process which was used to implement test automation scheme for functional testing of LTE core network MME element. In implementation of the test automation scheme agile development models and Robot Framework test automation tool were used. In the third part two alternative models are presented for integrating this test automation scheme as part of a continuous integration process.

As a result, the test automation scheme for functional testing was implemented. Almost all new functional level testing test cases can now be automated with this scheme. In addition, two models for integrating this scheme to be part of a wider continuous integration pipe were introduced. Also shift from usage of a traditional waterfall model to a new agile development based model in testing stated to be successful.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Tietotekniikan osasto
Antti Pohjonen

TESTI AUTOMAATIO JÄRJESTELMÄ LTE RUNKOVERKKO ELEMENTILLE

Diplomityö

2010

92 sivua, 27 kuvaa ja 3 taulukkoa

Tarkastajat: Professori Jari Porras
Diplomi-insinööri Henri Elemo

Hakusanat: Testi automaatio, LTE, ohjelmistotestaus, ketterä ohjelmistokehitys
Keywords: Test automation, LTE, Software testing, Agile development

Modernit kehittyneet tietoliikenneverkkolaitteet vaativat yhä enemmän ja kattavampaa testausta laadun varmistamiseksi. Kattavan testauksen tarvitsevat testitapaus määrät ovat nousseet huomattavasti ja manuaalisella testauksella tätä kasvanutta kysyntää ei pystytä enää tyydyttämään. Lisäksi uudet ketterät ohjelmistokehitysmenetelmät vaativat testien suorittamista jokaisen iteraatiokierroksen yhteydessä. Tämän takia laitevalmistajat ovat siirtyneet enenemissä määrin käyttämään testiautomaatiota riittävän kattavuuden ja laadun varmistamiseksi testauksessa.

Diplomityö on jaettu kolmeen osaan. Ensimmäisessä osassa esitellään soluverkkojen evoluutiota ja tutkitaan ohjelmistotestausta ja sen automatisointia sekä erilaisten ohjelmistokehitysmenetelmien vaikutusta testaamiseen. Toisessa osassa kuvataan prosessia jolla rakennettiin automaattinen toiminnallisuustestaus järjestelmä LTE runkoverkon MME elementille. Testiautomaatiojärjestelmän kehityksessä käytettiin ketteriä ohjelmistokehitysmenetelmiä ja Robot Framework testiautomaatio-ohjelmistoa. Kolmannessa osassa esitellään kaksi vaihtoehtoista mallia tämän järjestelmän liittämiseksi jatkuvaan integraatiojärjestelmään.

Työn tuloksena saatiin rakennettua automaattinen testaus järjestelmä. Lähes kaikki uudet toiminnallisuustestauksen testitapaukset voidaan automatisoida järjestelmän avulla. Lisäksi tehtiin kaksi vaihtoehtoista mallia järjestelmän integroimiseksi osaksi laajempaa jatkuvaa integrointi ympäristöä. Myös siirtyminen perinteisestä vesiputousmallista uuteen ketterän kehitysmenetelmän käyttöön testauksessa havaittiin onnistuneen.

Acknowledgements

This thesis work was carried out in the IP Control Department at Nokia Siemens Networks Oy in Finland. I would like to thank all people at Nokia Siemens Networks who have made this thesis possible.

I wish to express my gratitude to Professor Jari Porras for supervising this thesis and for his feedback. I also want to express my gratitude to Henri Elemo, for all the valuable advice at all stages of the work and for being the instructor of this thesis.

I would also like to thank the Robot Test Automation team and the rest of the Integration, Quality and Methods team members for their general feedback and support.

I also wish to thank Luisa Hurtado for revising the language of this thesis and for the encouraging support I got during this project.

Finally, I want to give special thanks for my family and friends for their support during my studies.

.

Espoo, 11 May 2010

Antti Pohjonen

Table of Contents

Abbreviations

1G	First Generation
2G	Second Generation
3G	Third Generation
3GPP	3G Partnership Project
4G	Fourth Generation
8-PSK	eight-Phase Shift Keying
AMPS	Advanced Mobile Phone System
ANSI	American National Standards Institute
API	Application Programming Interface
ATCA	Advanced Telecom Computing Architecture
ATDD	Acceptance Test Driven Development
CDMA	Code Division Multiple Access
CDPD	Cellular Digit Packet Data
CI	Continuous Integration
CPPU	Control Plane Processing Unit
DAD	Distributed Agile Development
D-AMPS	Digital Advanced Mobile Phone System
DSD	Distributed Software Development
DSSS	Direct-Sequence Spread Spectrum
E2E	End To End
EDGE	Enhanced Data rates of Global Evolution
ETSI	European Telecommunications Standards Institute
E-UTRAN	Evolved Universal Terrestrial Radio Access Network
FDD	Frequency-Division Duplexing
FDMA	Frequency Division Multiple Access
GGSN	Gateway GPRS Support Node
GMSK	Gaussian minimum shift keying
GPRS	General Packet Radio Services
GSD	Global Software Development
GSM	Global System for Mobile Communications

GTP	GPRS tunneling protocol
HSCSD	High Speed Circuit Switched Data
HSS	Home Subscriber Server
HTML	Hyper Text Markup Language
IEEE	Institute of Electrical and Electronics Engineers
IMT	International Mobile Telecommunications
IMT-DS	IMT Direct Spread
IMT-FT	IMT Frequency Time
IMT-MC	IMT Multicarrier
IMT-SC	IMT Single Carrier
IMT-TC	IMT Time Code
IPDU	IP Director Unit
IRC	Internet Relay Chat
ITU	International Telecommunications Union
JDC	Japanese Digital Cellular
LTE	Long Term Evolution
MCHU	Marker and Charging Unit
MIMO	Multiple-Input and Multiple-Output
MME	Mobility Management Entity
MTC	Main Test Component
NAS	Non-Access-Stratum
NGMN	Next Generation Mobile Networks
NMT	Nordic Mobile Telephone
NSN	Nokia Siemens Networks
OFDM	Orthogonal Frequency Division Multiplexing
OMU	Operational and Maintenance Unit
OS	Operating System
PCEF	Policy and Charging Enforcement Function
PCRF	Policy and Charging Rules Function
PDC	Personal Digital Cellular
PDE	Public Definition Environment
PDN-GW	Packet Data Network Gateway

PTC	Parallel Test Components
RAN	Radio Access Network
RF	Robot Framework
ROI	Return of Investment
S1AP	S1 Application Protocol
SAE-GW	System Architecture Evolution – Gateway
SC-FDMA	Single-Carrier Frequency Division Multiple Access
SCM	Software Configuration Management
SCTP	Stream Control Transmission Protocol
SGSN	Serving GPRS Support Node
S-GW	Serving Gateway
SMMU	Signaling and Mobility Management Unit
SMS	Short Messaging Service
SOAP	Simple Object Access Protocol
SUT	System Under Testing
SVN	Subversion
TACS	Total Access Communication System
TC-MTS	Methods for Testing and Specification Technical Committee
TDD	Time Division Duplexing
TDD	Test Driven Development
TDMA	Time Division Multiple Access
TD-SCDMA	Time Division - Synchronous Code Division Multiple Access
TR	Technical Report
TSV	Tab Separated Values
TTCN3	Testing and Test Control Notation Version 3
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
UWC	Universal Wireless Communication
W-CDMA	Wideband Code Division Multiple Access
WiMAX	Worldwide Interoperability for Microwave Access
XML	eXtensible Markup Language
XP	Extreme Programming

1. Introduction

The modern mobile telecommunication networks are complex and sophisticated systems. These networks are made up of many different elements which all communicate with each other with various interfaces and protocols. The amount of features and technologies these elements have to support is increasing rapidly with increasing demand and continuous development of mobile services around the world. The rapid increment of technologies makes it even more difficult to achieve adequate testing coverage for equipment manufacturers, because the amount of needed test cases is also increasing rapidly. The usage of new agile development models are adding pressure to execute all related test cases at least once in short development cycles. To address these challenges, test automation is raising its popularity for all levels of testing among the testing community. [1, 2, 3, 4]

This thesis describes the implementation of a test automation scheme for an Long Term Evolution (LTE) core network element. The core network element used as the system under test (SUT) was the mobility management entity (MME), which is responsible for session and mobility management and control plane traffic handling and security functions in an LTE network. The test automation scheme implemented in this thesis covers phases of automatic build commission to hardware, integration testing, fully automated functional testing and design of complete continuous integration system. Preceding phases before build compiling are introduced, but not covered in detail.

Development of MME core network element is carried out with agile development model and the scrum method. MME's development was done in multisite environments located in different geographical locations and time zones. Test automation scheme development was started using the same agile model and scrum method. After supporting functions started, the test automation scheme development method also changed to freer model, where supporting tasks had always the highest priority. Test automation development was carried out at one site in Espoo, Finland.

2. Evolution of cellular networks

This chapter describes the evolution of cellular networks and focuses on key technological reforms. Key points of technological changes are discussed in more detail, and mobile networks data transfer capacity is emphasized. This will give a general understanding to the reader of history, evolution and major standards of data transfer in cellular networks.

The first real cellular system was introduced in 1979 in Japan, but wider use of such networks started during the next decade. There were mobile networks even before that, but capacity and support for mobility was remarkably weaker, and hence those networks cannot be classified as cellular networks. The human need for constant movement and freedom from fixed locations has been the key factor for the success of cellular networks. [1, 2]

Since the start of the digital era with GSM technology demand for mobile service has grown tremendously, and after developing countries have started to invest in mobile technology, the demand has almost exploded. According to Goleniewski and Jarret in Telecommunications Essentials, *"The growth was so rapid that by 2002, we witnessed a major shift in network usage: For the first time in the history of telecommunications, the number of mobile subscribers exceeded the number of fixed lines. And that trend continues. According to the ITU, at year-end 2004, the total number of mobile subscribers was 1.75 billion, while the total number of fixed lines worldwide was 1.2 billion."* [1] Nowadays there are more than 4.2 billion mobile subscribers world wide according to a market study done in 2Q 2009. [5]

The driver of evolution in mobile networks has previously been the need for greater subscriber capacity per network until the third generation (3G). 3G networks were the first technological turning point in which individual subscriber demand for greater data transfer capacity was the driver, and the same driver is leading the way to next the generation of LTE and Worldwide Interoperability for Microwave Access

(WiMAX) mobile networks, along with service providers' need for better cost efficiency per transferred bit. [3, 6]

2.1. Analog systems in first generation networks

Cellular network first generation (1G) was based on analog transmission systems and was designed mainly for voice services. The first cellular network was launched in Japan in 1979, and launching continued through the 1980s in Europe and Americas. A new era of mobile networks was born. The variety of different technologies and standards was quite wide. The most important first generation standards were Nordic Mobile Telephone (NMT), Advanced Mobile Phone System (AMPS), Total Access Communication System (TACS) and Japanese TACS.[1,2]

NMT was invented and used at first in Nordic countries, but later on, was launched also in some southern and middle European countries. AMPS technology was used in the United States, Asian and Pacific regions. In the United Kingdom, Middle-East, Japan and some Asian regions, TACS was the prevailing technology. Also some country specific standards and technologies were used like C-Netz in Germany and Austria, Radicom 2000 and NMT-F in France and Comvik in Sweden. [2]

In first generation wireless data networks, there were two different key technologies; Cellular Digit Packet Data (CDPD) and Packet radio data networks. The latter one was designed only for data transfer, but CDPD used unused time slots of cellular networks. CDPD was originally designed to work over AMPS and could be used over any IP-based network. Packet radio data networks were built only for data transfer and its applications, such Short Messaging Service (SMS), email, dispatching etc. Peak speed of packet radio data networks were 19.2 Kbps, but normally rates were less than half of this peak performance. [2]

2.2. Digital era of 2G and 2.5G

The most remarkable generation shift in cellular networks was from first generation to the second. The 1G was implemented with analog radio transmission, and the 2G is based on digital radio transmission. The main reason for this shift was increased demand for capacity, which needed to be handled with more efficient radio transmission. In 2G, one frequency channel can be used simultaneously by multiple users. This is done by using channel access methods like Code Division Multiple Access (CDMA) or Time Division Multiple Access (TDMA), instead of using the capacity extravagant method of Frequency Division Multiple Access (FDMA), whose differences can be seen in *figure 1*. In these methods one frequency channel is divided either by code or time to achieve more efficient usage of that channel. [2]

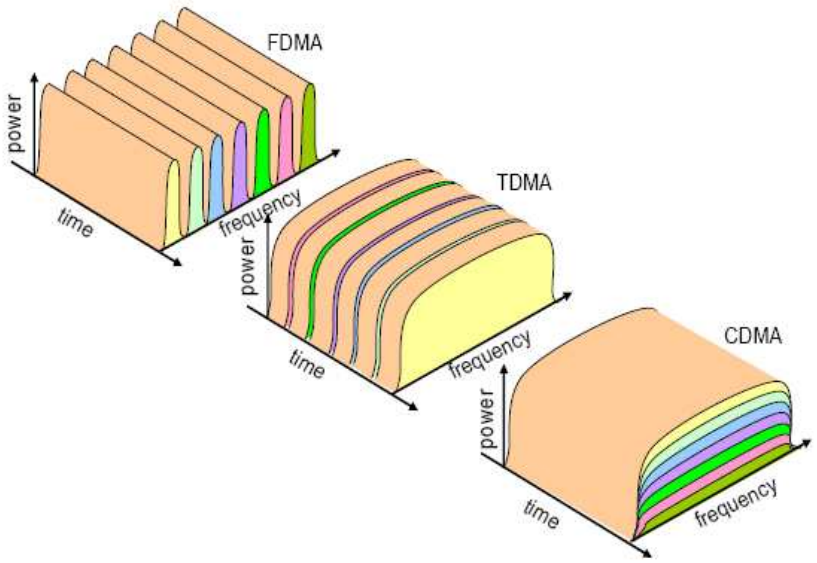

Figure 1: FDMA, TDMA and CDMA [7]

In 2G there are four main standards: GSM, Digital AMPS (D-AMPS), CDMA and Personal Digital Cellular (PDC).

GSM started as a pan-European standard, but was adopted and widely spread out to be true global technology. Currently it is the most used technology in mobile networks [5]. GSM technology uses TDMA with Frequency-Division Duplexing (FDD) in which downlink and uplink use a different frequency channel. Peak data rates in plain GSM technology could at first achieve only 9.6 Kbps, but later it increased to 14.4 Kbps. [1, 2]

D-AMPS also known as Universal Wireless Communication (UWC) or IS-136 and is mainly used in the Americas, Israel and some Asian countries. D-AMPS is also based on TDMA, but with Time Division Duplexing (TDD), where downlink and uplink uses the same frequency channel, allocated by using time slots. Basic IS-136 offers data transfer rates up to 30 Kbps, but with IS-136+ the range are from 43.2 Kbps to 64 Kbps. [2]

CDMA uses a different approach to dividing air interface than in TDMA based technologies, and it separates different transmission by code and not by timeslots. The first commercial CDMA technology is based on standard IS-95A and can offer 14.4Kbps peak data rates. CDMA is mostly used in networks located in United States and East Asian countries. PDC, formerly was known as Japanese Digital Cellular (JDC), but name was changed as an attempt to market the technology outside of Japan. This attempt failed, and PDC is commercially used only in Japan. PDC uses the same technological approach as D-AMPS and GSM with TDMA. PDC can offer circuit-switched data service rates up to 9.6 Kbps and, as a packet-switched data service, up to 28.8 Kbps. [2]

The line between 2G and 2.5G cellular networks is vague and cannot be defined strictly. 2.5G networks in general are upgraded 2G networks offering higher data transfer rates than basic 2G networks. In some cases those upgrades can be done only with software updates or minor radio interface changes. These upgrades are downward compatible, and so only subscribers who want advantages of newer technology have to update own devices. The general conception is that 2.5G cellular network should support at least one of following technologies; High Speed Circuit

Switched Data (HSCSD), General Packet Radio Services (GPRS), Enhanced Data rates of Global Evolution (EDGE) in GMS or D-AMPS networks and in CDMA networks technology, which is specified by IS-95B or CDMA2000 1xRTT. [1, 2]

HSCSD is the easiest way to boost GSM network capacity, and it needs only software updates to existing networks. Compared to plain GSM network, a HSCSD network uses different coding schemes and is able to use multiple sequential time slots per single user and this way boosts data transfer rates. This technological improvement does not help networks which are already congested, but instead can make them worse. A solution which needs real time communication HSCSD is preferred, because of the nature of circuit switched connection. HSCSD data transfer rates range from 9.6 Kbps up to 57.6 Kbps with using time slot aggregation and can be raised up to 100 Kbps, if channel aggregation is used. [1]

GPRS is technology which requires a few new main components to the core network and updates to other elements as well. The new core components are Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN). SGSN handles control signaling to user devices and data routing inside SGSN's serving area. GGSN handles GPRS roaming to other networks and is the gateway between public networks like Internet and GPRS network. GPRS technology is a packet-switched solution and can achieve a maximum of 115 Kbps peak data rate. [1, 2, 8]

EDGE is a third option to upgrade TDMA based cellular networks. EDGE technology takes advantage of an improved modulation scheme which in most cases can be achieved only by software updates. In EDGE, data transmission is boosted by using eight-Phase Shift Keying (8-PSK) instead of basic Gaussian minimum shift keying (GMSK). This will improve transmission rates up to threefold. [1, 2]

CDMA networks also have some updates to speed up data transfer. These upgrades are IS-95B, CDMA2000 1xRTT or Qualcomm's proprietary solution of High Data Rate, which is also known as 1x Evolved Data Optimized and is a nonproprietary solution of this technology. This can boost data transfer rates up to 2.4 Mbps, which

is similar as in early implementations of 3G, although these upgrades are still considered as 2.5G technology. [1]

2.3. 3G The beginning of mobile broadband

Third generation (3G) cellular networks design started soon after second generation networks were commercially available. European Telecommunications Standards Institute (ETSI) and key industry players were among the first to start studies. A few key design principles were truly of a global standard regarding high speed mobile broadband data and voice services. New services like high quality voice, mobile Internet access, videoconferencing, streaming video, content rich email, etc. created a huge demand for mobile data transfer capacity, and 3G is designed to meet these demands. The earliest 3G networks offered only a little better or the same transfer rates as most evolved 2.5G systems, but technologically there is a clear difference. [2]

3G has two major standards; Universal Mobile Telecommunications System (UMTS) and CDMA2000. Also there is country specific standard Time Division - Synchronous Code Division Multiple Access (TD-SCDMA) in China. For UMTS ETSI, organizational members and industry leader manufacturers founded the 3G Partnership Project (3GPP) in 1998 which functions under ETSI. A similar partnership program was founded to coordinate CDMA2000 development by the American National Standards Institute (ANSI) and organizational members called 3GPP2. [1, 2, 9, 11]

The International Telecommunications Union (ITU) has defined an umbrella specification International Mobile Telecommunications (IMT) 2000 for 3G mobile networks. It was meant to be truly global, but for political and technical reasons it was infeasible. The specification defines 5 different sub specifications; IMT Direct Spread (IMT-DS), Multicarrier (IMT-MC), Time Code (IMT-TC), Single Carrier

(IMT-SC) and Frequency Time (IMT-FT). All current 3G standards fit under those specifications. [1, 10]

UMTS is based on Wideband Code Division Multiple Access (W-CDMA), which is an alias for this standard, and it uses FDD or TDD for radio transmission. The UMTS was at first the 3G standard which acted as an evolution path for GSM systems, but later on the UWC consortium also took it as an evolution path for North American TDMA based systems like D-AMPS. [1, 2]

The CDMA2000 standard family is 3GPP2's answer to the CDMA network's evolution towards 3G. It was the first 3G technology commercially deployed. IS-95 High Data Rates (HDR), Qualcomm's propriety technology, is a base for CDMA2000 and is optimized for IP packets and Internet access. CDMA2000 uses multicarrier TDD for radio transmission. As well as UMTS also CDMA2000 is also based on Wideband-CDMA (W-CDMA) technology, but it is not interoperable with UMTS. [2, 11]

TD-SCDMA is a standard which is used mainly in the Chinese market. It is WCDMA technology and uses Direct-Sequence Spread Spectrum (DSSS) and TDD in radio transmission. [2]

All current 3G technologies are still evolving. Already many upgrades and enhancements have been made, and new ones will be coming before next generation networks are commercially available. The most significant technological enhancements currently are High Speed Packet Access (HSPA) for UMTS networks and CDMA2000 3X for CDMA2000 networks. [1, 2, 9, 11]

3. LTE network architecture

LTE is the name for 3GPP's fourth generation (4G) cellular network project. From a technological point of view, it is not really 4G technology, because it does not fully comply with ITU's IMT Advanced, which is considered to be the real umbrella standard for 4G systems [10, 12]. In this chapter key points of LTE network architecture and its structure are discussed. The MME core network element is covered in more detail, because it is the device used as the SUT in this thesis. The motivation and reasoning behind LTE technology are also presented. This will help the reader to get a general understanding of technologies related to this thesis and to introduce the element used in testing in more detail.

The motivation for developing the new technology release called LTE, which was initiated in 2004, can be summarized with six key points. First was the need to ensure the continuity of competitiveness of the 3G system for the future. This means that there had to be commercially competitive new technology available from the 3GPP group to support industrial manufacturer members to hold their market share against other rival technologies. Second was user demand for higher data rates and quality of service which arises from demand for increased bandwidth for new services like video streaming, virtual working, online navigation etc. Third was the technological shift to use an entirely packet switch optimized system. Continued demand for cost reduction in investments and for operational costs was a key driver for operators and the fourth point. Fifth was the demand for low complexity, meaning that network architecture had to be simplified. Sixth was to avoid unnecessary fragmentation of technologies for paired and unpaired band operation. [12, 13, 14]

The technological requirements for LTE were finalized in June 2005 by the next generation mobile networks (NGMN) alliance of network operators. These requirements are defined to ensure radio access competitiveness for the next decade. The highlights of requirements are reduction of delays, increased user data rates and cell-edge bit-rate, reduced cost per bit by implying improved spectral efficiency and

greater flexibility of spectrum usage, simplified network architecture, seamless mobility and reasonable power consumption for the mobile terminal. Reduction of delays is meant to happen in terms of connection establishment and transmission latency. The spectrum usage goal is meant for implementation in both new and pre-existing bands. The requirement of seamless mobility refers to different radio-access technologies. To achieve these requirements, both radio interface and the radio network architecture needed to be redesigned. [12, 13, 14]

As 3GPP's technical report (TR) 25.913 declares, LTE's high spectral efficiency is based on usage orthogonal frequency division multiplexing (OFDM) in downlink and single-carrier frequency division multiple access (SC-FDMA) in uplink. These channel access methods are robust against multipath interference, and advanced techniques, such as multiple-input and multiple-output (MIMO) or frequency domain channel-dependent scheduling, can be used. SC-FDMA also provides a low peak-to-average ratio which enables mobile terminals to transmit data power efficiently. LTE have support variable bandwidths like 1.4, 3, 5, 10, 15 and 20MHz. Simple architecture is achieved by using evolved NodeB (eNodeB) as the only evolved universal terrestrial radio access network (E-UTRAN) node in radio network and reduced number of radio access network (RAN) interfaces S1-MME/U for MME and system architecture evolution – gateway (SAE-GW) and X2 between two eNodeBs. [15]

According to the same TR 25.913, the LTE system should fulfill the following key requirements:

For peak data rate:

- Instantaneous downlink peak data rate of 100 Mb/s within a 20 MHz downlink spectrum allocation (5bps/Hz)
- Instantaneous uplink peak data rate of 50 Mb/s (2.5 bps/Hz) within a 20MHz uplink spectrum allocation

For control-plane latency

- Transition time of less than 100 ms from a camped state to an active state
- Transition time of less than 50 ms between a dormant state and an active state

For control-plane capacity

- At least 200 users per cell should be supported in the active state for spectrum allocations up to 5 MHz

For user-plane latency

- Less than 5 ms in unload condition for small IP packet

For mobility and coverage

- E-UTRAN should be optimized for low mobile speed from 0 to 15 km/h
- Higher mobile speed between 15 and 120 km/h should be supported with high performance
- Mobility across the cellular network shall be maintained at speeds from 120 km/h to 350 km/h (or even up to 500 km/h, depending on the frequency band)
- The throughput, spectrum efficiency and mobility targets above should be met for 5 km cells, and with a slight degradation, for 30 km cells. Cell ranges up to 100 km should not be precluded.

The 3GPP release 8, simplified LTE non-roaming network architecture, can be seen in *figure 2*. LTE non-roaming architecture has E-UTRAN and user equipment (UE) as radio network access components. In evolved packet core (EPC) network part have SGSN, MME, and home subscriber server (HSS) which act as subscriber database, serving gateway (S-GW) and packet data network gateway (PDN-GW) and policy and charging rules function (PCRF), which has policy management rules for subscribers and applications etc.[16]

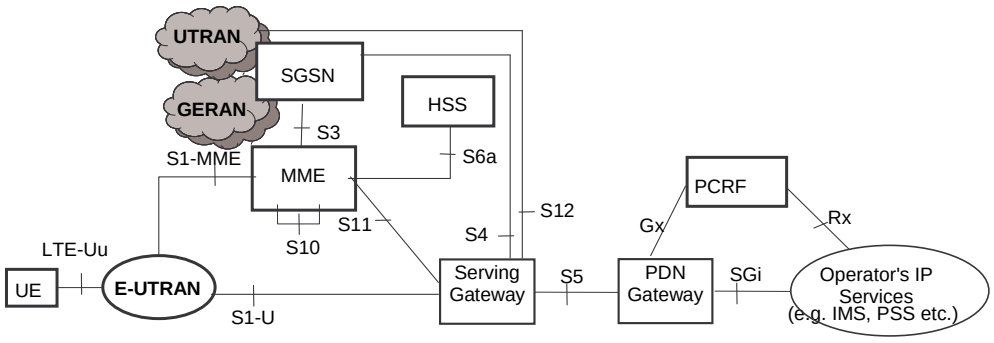

Figure 2: LTE network architecture [16]

3GPP technical specification 23.401 also defines interfaces and reference points which are used in the LTE system:

S1-MME: Reference point for the control plane protocol between E-UTRAN and MME.

S1-U: Reference point between E-UTRAN and Serving GW for the per bearer user plane tunneling and inter eNodeB path switching during handover.

S3: Enables user and bearer information exchange for inter 3GPP access network mobility in idle and/or active state.

S5: Provides user plane tunneling and tunnel management between Serving GW and PDN GW. It is used for Serving GW relocation due to UE mobility and if the Serving GW needs to connect to a non-collocated PDN GW for the required PDN connectivity.

S6a: Enables transfer of subscription and authentication data for authenticating/authorizing user access to the evolved system between MME and HSS.

Gx: Provides transfer of policy and charging rules from PCRF to Policy and Charging Enforcement Function (PCEF) in the PDN GW.

S10: Reference point between MMEs for MME relocation and MME to MME information transfer.

S11: Reference point between MME and Serving GW.

SGi: Reference point between the PDN GW and the packet data network. A packet data network may be an operator external public or private packet data network or an intra operator packet data network. [16]

3.1. MME network element

A MME network element in an EPC network acts as a session and mobility management node. MME's main responsibilities are subscriber's authentication and authorization, control plane traffic handling, security functions and session and mobility management in LTE radio network and between other 3GPP 2G/3G access network or non-3GPP radio networks and LTE radio networks. MME is a dedicated

control plane core network element, and all user plane traffic is directly handled between eNodeB and S-GW [17]. Due to the flat architecture of LTE radio networks, all eNodeBs are directly connected to MME, which require more mobility management transaction with active mobile subscribers [14].

Session and mobility management include various functions, such as tracking area list management, PDN GW and Serving GW selection, roaming by signaling towards home HSS, UE reachability procedures and bearer management functions including dedicated bearer establishment [14]. Besides those functions, MME also handles non-access-stratum (NAS) signaling and security, SGSN selection for handovers to 2G or 3G 3GPP access networks and lawful interception signaling traffic [16].

Signaling between MME and E-UTRAN is done with S1 application protocol (S1AP) which is based on stream control transmission protocol (SCTP) [14]. S3 or Gn is the interface between MME and SGSN, where S3 is SGSN which comply with 3GPP technical specifications Release 8 or newer, and Gn is for Release 7 or earlier. S10 interface is for control plane traffic between two MMEs and S6a between HSS and MME for getting subscriber information [16]. S11 interface is meant for the control plane between MME and S-GW for handling bearer related information. All MME interfaces S3/Gn, S6a and S10 are based on SCTP, except S11, which is based on GPRS tunneling protocol (GTP) [16, 18].

NSN MME core network element

Nokia Siemens Networks implementation of MME is based on advanced telecom computing architecture (ATCA) hardware which can be seen in *figure 3* [17]. ATCA is hardware made with specifications of PCI Industrial Computer Manufacturers Group which is a consortium of over 250 companies. The purpose of this consortium is to make high-performance telecommunications, military and industrial computing applications [19]. By using ATCA hardware as a base, it gives an advantage for using high end carrier grade equipment. With ATCA, new technology updates like computing blades and interface options are available along with industry evolution,

meaning that new technologies can be brought to market faster, and component lifetime in operation will be longer. An ATCA shelf has 16 slots for computer units from which 2 slots are reserved for HUB blades for internal switching and communication purposes [19].

Figure 3: MME ATCA hardware [17]

NSN MME has five different key functional units; operational and maintenance unit (OMU), marker and charging unit (MCHU), IP director unit (IPDU), signaling and mobility management unit (SMMU) and control plane processing unit (CPPU) [17]. IPDU is responsible for connectivity and load balancing, MCHU offers statistics functions, SMMU handles subscriber database and state-based mobility management functions and CPPU is responsible for transaction-based mobility management [17]. In *figure* 4 this assembly is illustrated. SMMU also provides S6a interface to HSS and CPPU S1, S11 and S3/Gn interfaces. OMU and MCHU are redundant 2N units,

meaning there have to be N amount of pairs of both working and spare units. IPDU and SMMU are N+1 which means there has to be at least one working unit and exactly one spare. CPPU is N+ unit meaning one unit should be in a working state and the rest of CPPU units can be either in working or spare states [17].

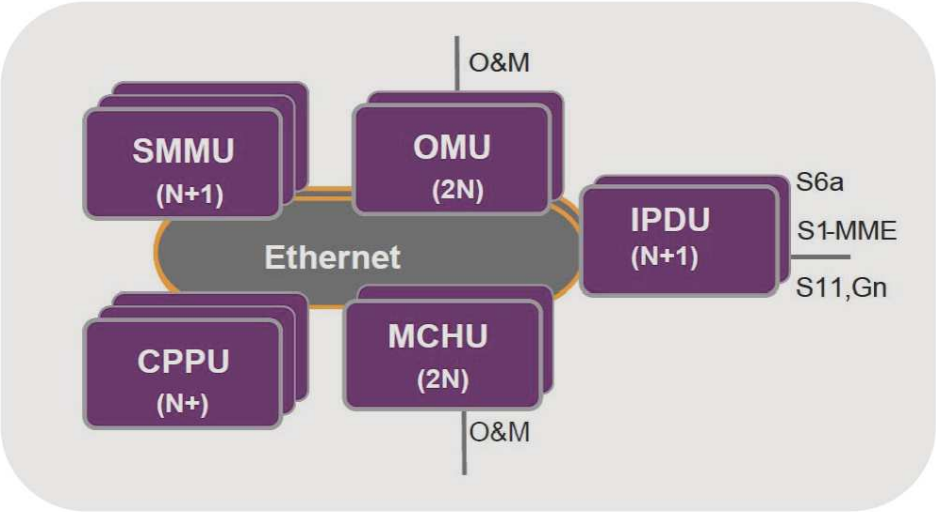

Figure 4: NSN MME units [17]

4. Software Testing and Test Automation

Software testing is a part of development where, systematically, defects and faulty functionality are searched for improving software quality and reliability. This is normally done via verification and validation with appropriate test tools and methods. This chapter describes software testing and the benefits of test automation. Also, different levels of testing are introduced and described in more detail. This context helps the reader to understand the complexity and requirements for implementing similar test automation schemes as described in thesis.

4.1. Definition of software testing

Software testing is a process where one tries to search and find defects, errors, faults or false functionality. It is not a series of indefinite attempts or vague experiments to make software crash or misbehave, but rather it is a systematic, carefully designed, well defined and executed process from which results are analyzed and reported [4, 21, 20]. The only form of testing that does not follow this formula is exploratory testing, but it is usually backwards traceable and well documented [22].

There are three different methods in software testing; white box testing, black box testing and gray box testing. In white box testing the test engineer has access and knowledge of the used data structures, code and algorithms used inside the software. The test engineer usually also has the possibility to access and test software with a wide array of tools and procedures like unit tests, module tests, internal interfaces, internal logs etc. In black box testing the test engineer's knowledge of software is limited to the specification level. One can only use external interfaces to test software and rely on information that the specification gives. In this method, software is seen as a black box which will reply on input with some output defined in the specification. Gray box testing is a combination of the two preceding methods. The test engineer has access and knowledge of internal data structures, code and

algorithms, but one will use external interfaces to test the software. This method is usually used in integration testing of software. [4]

In software testing software's behavioral deviation from specification is referred to as a defect, bug, fault or error. This derives to the conclusion, that without a specification, no exact testing can be done, and because any specification cannot be all-inclusive, not all found misbehaviors are considered defects, but instead features from developer's point of view. The customer, however, can think otherwise [20]. A general conception is that when defect occurs, it leads to a fault which can cause a failure in software. Failure is a state which is shown as misbehavior to other parts within software. For example, memory leak is a fault which can lead to a software crash, which is a failure. Not all defects are faults, and not all faults lead to failure. An error is thought to be a human misbehavior leading to defect, but all terms that describe defects are usually used as synonyms and interchangeably [21]. Also, the definition of testing and debugging are usually mixed and are used as synonyms, but the purpose of testing is to find misbehavior and defects from the software, whereas debugging is used to describe the process where programming faults are traced from the actual code [20].

Software testing is a combination of two processes, verification and validation. The meanings of these processes are quite often mixed and used interchangeably. The definition of verification according to the Institute of Electrical and Electronics Engineers (IEEE) standards, is the process of evaluating that software meets requirements that where imposed at the beginning of that phase. Software has to meet requirements for correctness, completeness, consistency and accuracy during its whole life cycle, from acquisition to maintenance. After implementation, those requirements, including specification and design documents and implemented code are checked to be consistent. Checking of code can be done only with code review, and no actual execution of software is needed. The definition of validation, according to IEEE, is the process of evaluating that the software meets the intended use and user requirements [23]. This means that software is usually executed against a set of test cases and as a general concept, this is thought to be the actual testing process.

4.2. Testing coverage and risk management

It is not feasible to perform an all-inclusive testing which finds all underlying defects even in the simplest of software. This is due to the large number of possible inputs, outputs or paths inside the software, and the specification of the software is usually subjective, meaning that the customer and developer don't usually see everything in the same way [4]. To fulfill all possible test scenarios would require a tremendous amount of test cases and through an infeasible number of resources. For these reasons are the cause why all-inclusive testing can be considered an impossible challenge. [4]

As mentioned earlier, not all defects lead to failure, and many defects are not even noticed during a software's whole life cycle [2]. Other defects can be just cosmetic or annoying, but do not cause any harm. Then there are those defects that most probably will lead to misbehavior or even failure of software and must be considered critical [2]. Another thing one must consider besides the amount of resources invested on testing is the phase in which testing is done. *Table 1* shows how much more it would cost to fix a defect at later phase in development than if it were discovered and fixed at some earlier phase [24]. It is a general conception that defects found and fixed during an early phase of development can save a lot of effort and resources later on [21].

Defect in	Defect discovered in				
	Requirements	Architecture	Implementation	System Test	Post-Release
Requirements	1x	3x	5-10x	10x	10-100x
Architecture		1x	10x	15x	25-100x
Implementation			1x	10x	10-25x

Table 1: Cost of finding defect [24]

Overall in testing the key question is; what is the optimal amount of testing; the point where one test enough to catch all critical defects that can affect on quality, but does not waste resource on economically infeasible over testing [21]. With less testing, one will safe resources, but will most certainly let some defects through [4]. This problem is illustrated in *figure 5* in which the horizontal axel is test coverage and the vertical axis represents quantity [4].

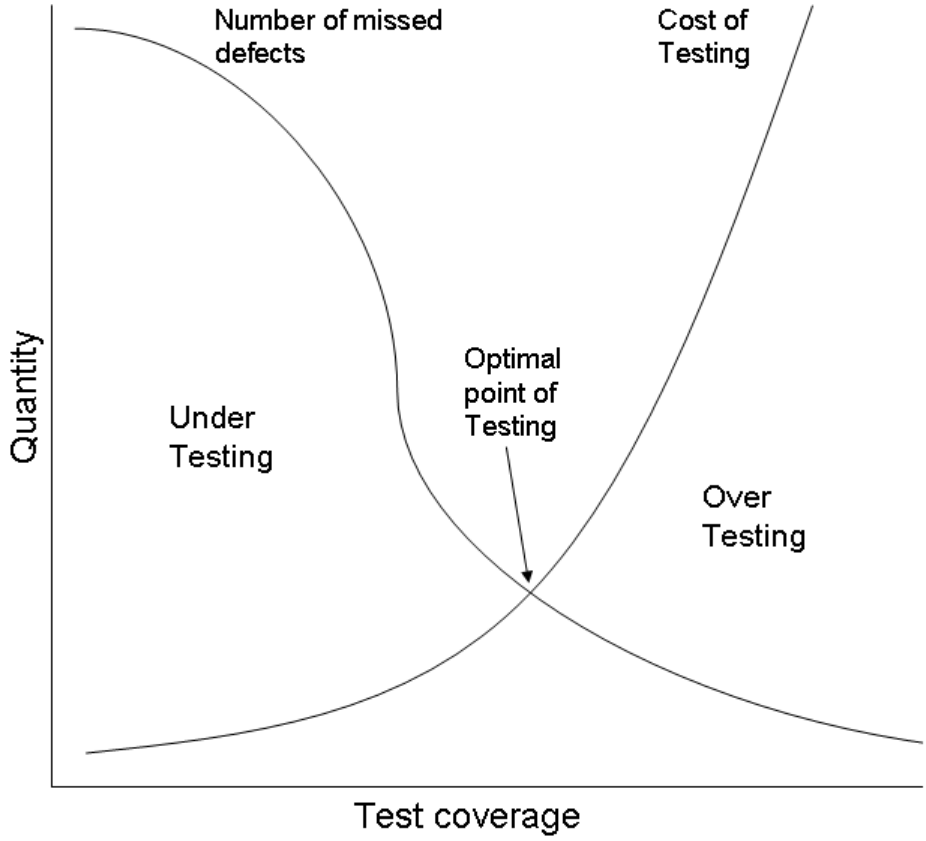

Figure 5: Optimal amount of testing [4]

Commonly the concepts of quality and reliability are mixed and thought to have the same meaning, but actually reliability is a subset of quality and only describes how robust, dependable and stable a software is [4]. Quality, instead, defines a larger set of aspects like the number of features, correctness of requirements, degree of excellence and price etc [4].

4.3. Testing levels and possibility for test automation

The different levels of testing are unit, module, integration, functional, system and acceptance testing, and all are classified as functional testing [20]. There has been developed a special v-model for testing as shown in *figure 6*, in which on the left side is the design and planning phases and the corresponding testing levels on the right side. The number of different phases varies depending how those are classified. Testing starts from the bottom and goes up. The model also indicates the cost of fixing a defect, which also increases depending on the level where it is found. Each testing level usually has its own coverage, purpose, tools and method. One has to also set also entry and exit criteria for each testing level which defines when software is robust and mature enough for a new level and when to terminate testing [21]. In unit or module testing the white box testing method is used, integration testing is done with the gray box method and functional, system and proceeding testing is carried out as the black box testing. There is also non-functional testing including performance, usability, stability and security testing which are normally executed after functional or system testing has been successfully executed. In non-functional testing, the method is usually gray or black box testing depending on the tools that are used. [21, 20]

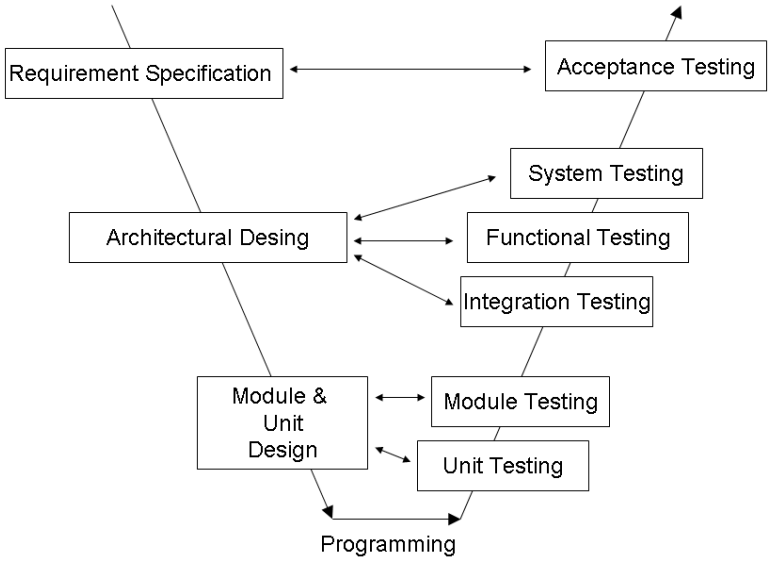

Figure 6: V-model of testing

Unit testing is mainly conducted to test functionality of the lowest level in code. Usually tests are targeted to function or method level in the smallest possible portions and testing all possible lines and different paths inside one module or class. Unit test results nowadays contain information about code coverage which tells the percentage of how many lines were accessed during tests. Tests are usually written by programmers themselves with the white box testing method to gain information about function or method correctness. Tests are usually carried out by executing part of the code in a simulated environment in test beds. This level of testing is usually can be fully automated and needs no human interaction. Results of tests are normally compared to unit design specifications. [21, 20]

Module testing is usually considered the same as unit testing, because methods and tools are the same. The main differences are in goal, testing scope and requirements for test execution. The coverage of module testing, measured in lines, is less meaningful, and right functionality of module or class is emphasized more than in unit testing [20]. The scope is to ensure correctness of functionality and the external interface of the module. When executing a module test, one usually needs to use mocked versions of other modules interacting with the module under test. Those are commonly referred to as test stubs or test drivers [21]. As with unit testing, module testing can also be executed in simulated environment in test beds and can be fully automated [20].

Integration testing is the first testing phase where modules are tested together to ensure proper functionality of interfaces and cooperation of modules. There are usually three practices to carry out integration; big bang, bottom up and top down. In big bang all changed modules are combined in a “big bang” and tested. If software is goes through compiling phase and starts up this practice is probably the fastest way to find and fix interface and interoperability defects and issues [20]. In bottom up integration, testing is built up by adding the lowest level module first and continue adding next levels until reaching highest level module [20]. The top down practice uses just the opposite way of doing integration than bottom up. Methods used in integration varies from high level white box testing to gray box testing and normally

with bigger software programs, the gray box testing method is dominant [21]. Full automatization of this phase is infeasible at the beginning of development, because many defects found in an early state of integration testing are causing difficulties to even start or keep software running. Without software running at least in a moderately stable way, it can be impossible or at least infeasible to collect the necessary logs. A sudden crash of software usually will leave incomplete logs, which often are useless. When software achieves a sufficient state of robustness and stability, integration testing can be also fully automated. In integration testing, at least the basic functionality of software should be tested [20].

Functional testing can be considered to be part of system testing or preliminary testing before it. Functional testing is mainly done as gray box testing to verify correctness of systems features and proper functionality. System testing is carried out normally with black box testing, where the knowledge and information of internal functionality is limited [20]. Hence the functional testing level has the opportunity to observe internal logs and monitoring. This phase is the first phase where all features of a software are tested and checked that they correspond to the architectural design and specifications. This all-inclusive testing is called regression testing, and its purpose is to clarify that new functionality has not broken down old functionality. With good test environment design and planning, almost all test cases can be automated. There are always test cases which require infeasible requirements like sudden loss of power or are used only once. Automating these types of test cases is impractical [20].

The system testing phase is where software is tested against the highest levels of architectural design and system specifications and testing is carried out using the black box method. Usually in this phase, real hardware is used along with other real system elements. In the telecommunication industry network verification belongs under system testing, where almost all network elements are real hardware [4]. The automation level of testing is the same as in functional testing but even more feasible, because no internal logs or monitoring needs to be collected. Defects and

faults found in this level are usually caught through network analyzers or external logs of hardware equipment [4].

The acceptance testing is done through cooperation with a customer against the user requirement specification. One part of this testing can also be usability testing where the user interface is tested. The purpose of this level is to make sure product is compliant what the customer ordered. This is done also with the black box testing, usually by using automated or semi-automatic test cases, but with execution is initiated manually. [20]

Non-functional testing like performance, security and stability testing, normally is carried out simultaneously with system testing, and some consider it to be part of it [20]. However, the tools and scope are quite different in non-functional testing than in system testing. Performance testing usually cannot be executed using the same tools as functional testing, and the scope of functionality under test is normally many times narrower. In stability testing, the scope is in causing a heavy load into the system and analyzing the system capability to handle it [21].

4.4. Test automation versus manual testing

In modern software development, the main question is not as unambiguous as should one use test automation or manual testing? Rather one should think of the level of automation to be used and which levels of testing are feasible to automate [4]. There are many advantages of test automation, such as speed, accuracy, reproducibility, relentlessness, automated reports etc., however maintenance load of test cases, more complex designs and implementation of test cases can make test automation unfeasible to be used in every situation [22]. One should always consider the return of investment (ROI) when determining whether to automate some testing level and which level of automation to use [3].

The main advantages of test automation are almost always taken for granted when speaking on this subject. The following list explains these advantages in more detail.

- Speed is maybe the most important single reason to consider when deciding the implementation of test automation. In the time a human takes for writing a 25 digit command and pressing enter, a modern computer can do it hundreds, thousands or even millions of times. Result analysis of a single line response like prompt or *command executed* by a human can take about one second and by computer only milliseconds. This kind of advantage is emphasized in regression testing which can contain millions commands and responses. [3, 4]
- Accuracy, a computer will always execute the test case as exactly as it is written. It will not make mistakes. If an error does occur, it is either written in the test case or the result of some external disturbance. [4, 22]
- Reproducibility can be considered to be accuracy plus timing. Even if the test engineer would not make mistakes timing can be the significant factor when trying to reproduce a defect which is caused by some timing issue. Of course, variable timing can be also seen as an advantage in testing, but it should be controlled and not caused by contingency. [4, 22]
- Relentlessness is one of machines' best advantage over humans and is defined as accuracy combined with reproducibility plus stubbornness. A machine will never give up or get tired and keeps performing its ordered tasks until it is finished or it breaks apart. [4, 22]
- Automated reports and documentation are always taken as by products of test automation, but provide a great advantage, since some cases, they can reveal what will or has been actually tested vs. the specification documentation. Modern testing frameworks even recommend and encourage making test cases as human readable as possible [25]. After execution, test automation tools usually will generate reports and logs and make it easier to prove what was actually tested and what were the results. [4, 22, 26]
- Provides resources that are needed to test large test contexts simultaneously like thousands of connections to some server application [4]. This can be

extremely difficult or impossible to manage manually and to successfully open sufficient amount of connections [4]. These repetitive tasks call for test automation.

- Efficiency. A test engineer's talent and capability is not fully used if one will be tied up in following test executions. Test automation frees up test engineers to do more valuable tasks, like test case design and result analysis etc. It also improves motivation and job meaningfulness, if routine tasks can be left for machines. [3, 4, 22]

Test automation is less advantageous with the more complex test case designs, which will need different points of view for testing, need for possible new investment for test automation equipment, and competence to operate new systems. Also maintenance of test cases is one field that needs more focus than with the manual testing, where maintenance is usually done within the testing routine [22].

The levels of automation can be divided into three distinct categories: full automation, semi-automatic and manual testing. In the full automation level, all procedures are automated, except for thedeeper analysis of failed cases which is done manually. With a successful outcome, the full automation level process does not require any human interaction. The setup, test execution, and teardown phases are all automated, and starting, ending and possible next phase triggering is handled by the testing system. In semi-automatic testing, at least one of the testing phases of the setup, execution or teardown must be done manually if too complex or fluctuating to be automated. The manual testing category contains test cases which are not in any way feasible to be automated. One-off, usability and exploratory tests are good examples of this category. One-off test cases are tests that are intended to be executed only once or very few times. This is because the functionality or feature under test is only a temporary solution and will be replaced or erased soon resulting in an ROI too low to justify its automation. Usability tests usually are included in this category, because user interface tests are difficult to automate due to results being based on empirical factors and can be extremely hard to describe in any deterministic way. User interface is a part of software which is constantly changing, with little

variation. Exploratory test descriptions already reveal reasons why those test are not practical to automate, based on exploring and testing software and designed tests not already applicable. If any defects are found through exploratory testing, there can be efforts to make automated test cases for those specific cases, but all preceding exploratory tests and results must be carefully documented in detail for reproducibility reasons. [22, 26, 27]

In designing a test automation strategy there is a good model, having a shape opposite the V-model, which is used to show the importance of ROI and the need of test automation in certain testing levels. The model is called Mike Cohn's Test Automation Pyramid [22] shown in *figure 7***.** The pyramid can be used as a guide on where to invest test automation resources. [22]

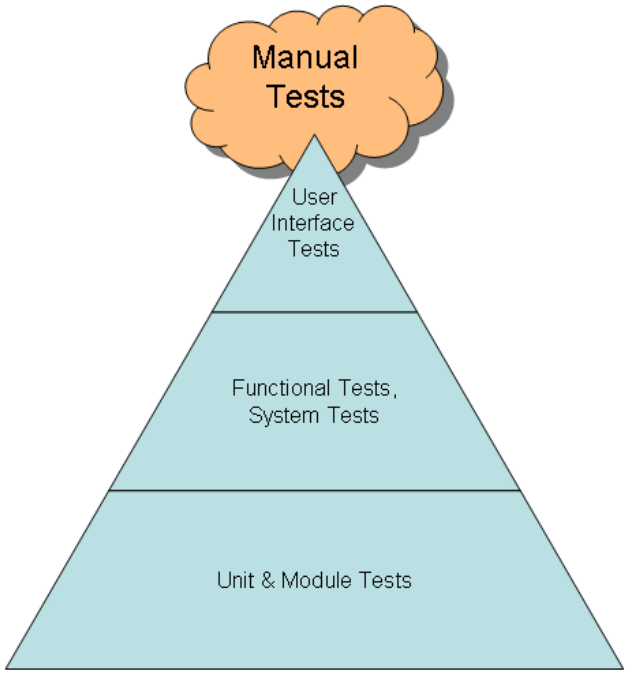

Figure 7: Mike Cohn's Test Automation Pyramid

As from the figure above, it can be observed that the unit and module tests are the base of test automation pyramid. After an acceptable level of automation is achieved at a certain level, one can start to automate next level test cases. Unit and module level tests should be automated as a first thing, and as much as feasibly possible.

Normally, tests are written with the same language as the software and by the programmers themselves. The next level, functional and system tests, is also important, but if the preceding base of unit and module tests are leaking, this level's number of caught defects will skyrocket, and it will be come inefficient. This level should be only used to catch architectural defects. The user interface level should be the last level in which to invest resources, because ROI is the lowest at that level. Manual tests are shown as a cloud, because there is always possibility that all test cases are not possible and feasible to automate. [22, 28]

5. Software development model's influence on testing

This chapter presents different software development models and their influences on testing. Also, the effects of distributed software development (DSD) are discussed, and the continuous integration (CI) process is briefly described [29]. Firstly, the most used development models are presented with a short description of each and their influence on testing requirements considered. Then distributed software development and its subclass distributed agile development (DAD) are presented [30]. Finally a short introduction to the continuous integration process and its requirements for testing is analyzed.

Software development is a process in which software is created with certain structured way. For handling this software development process there are several readymade models to describe the tasks and actions needed in each different phase of development. Those models can be divided in two categories; traditional development models and agile development models. There is also third category called iterative models, but usually those have same kind of workflow as traditional models, but with a smaller cycle content. In traditional development models, all steps are usually predefined before starting each step, and developers just follow the specifications of each step, progressing in linear mode from each step to next. In agile development, the only requirements are predefined, and developers make software in iterative steps which can be done simultaneously in parallel mode. These two approaches set quite different requirements for testing. In traditional models, testing is done at the end of the development cycle, but in agile models, development can start by doing acceptance tests first and executing those tests throughout development until they are pass, signifying the end of the development cycle. [4, 22, 29, 30]

5.1. Traditional development models

Traditional development models are normally based on the idea of sequential and well defined upfront design. This means starting a project by defining requirements and making requirements specification and guide lines for each step or phase of the project. After this step architects start the overall design of the different phases and architectural design. All required tasks and steps are carefully designed and documented upfront, so that in the implementation phase all developers can just concentrate on making their own tasks. Testing, including verification and validation, is usually done as the last step of the project with testing plan made during requirements specification. This means that if there is a defect found during testing or other needs for change after requirements specification, the whole process has to be started from the point where that change is needed and proceeding development phases have to be re-done. Hence, traditional development models do not welcome any changes during the development process, after the initial requirement specification is done. Any major changes are usually transferred to the next release. Four most frequently used traditional development models are Big Bang, Code and Fix, Waterfall and Spiral model. There are many more, but those are usually just variations of these four. [4, 20, 21]

Big Bang Development Model

The Big Bang model is not really any structured model, but it has to be mentioned, because many of nowadays famous products and services have been initially started with a similar approach. In the Big Bang model, instead of having requirements, even hint of specifications or fixed schedule, the customer only has an idea and the resources to start doing it. There is no deadline or even guarantee that the project will ever get anything ready. The idea behind this model is the same as in the dominant theory of the creation of universe, with a huge amount of energy and resources that together will create something special. Sometimes this model works, but there is a similar chance it will lead to nothing. Testing in this model is just finding defects by

using the product as your testing specification. In many cases, defects that you find are just meant to be told to the customer, not to be fixed. [4]

Code and Fix Development Model

Code and fix model is the next step from the Big Bang model. There usually is some informal requirement specification, but not a very well defined one. This is a model where many projects fall into, if project fails to follow some more specific model. The model suits small and light projects, where the only goal is to make a prototype, proof of concept or a demo. This three phase development model can be seen in *figure 8*. First developers try to make the product by following specifications. In the second phase follows some kind of testing of the product which is usually not a formal structured specification based process, but instead, more like exploratory testing. If testing results are satisfying for the customer project ends and the product is released, but if defects are found, the project goes one step back into the development phase and tries to fix the defect. This fixing, developing and testing cycle can be carried on until the product satisfies the customer, resources end or somebody has the courage to blow the whistle and end it. There isn't usually any predefined strict deadline for these projects or it has been exceeded a long time ago and project just tries to complete the assignment using this model. There is not any separate testing phase in this model, but testing is carried out in the development cycle where all three steps programming, testing and redesign are constantly repeated until the project is over. [4, 20]

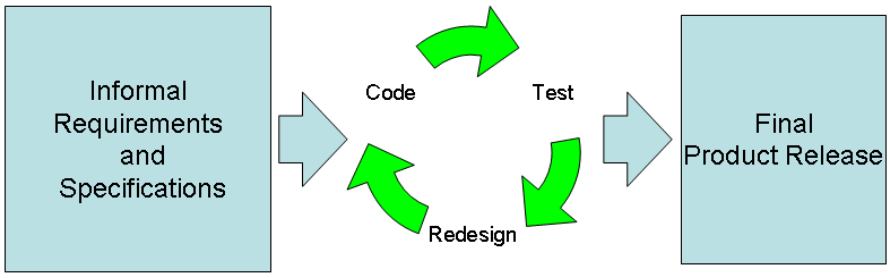

Figure 8: Code and Fix development model

Waterfall Development Model

Waterfall is the most famous of the traditional development models. It is a deterministic process that consists of different discrete steps. It is used in the development process from the tiniest programs to very large projects, because it is simple, sensible and scalable. *Figure 9* shows the usually used steps of the modern Waterfall model from requirements to product release. The project that follows the Waterfall model has to do every complete step until they can proceed to the next step. At the end of this step, the project should make sure that all required tasks of that step are carried out exactly and without loose ends. Moving from the preceding step to the next makes this model look like a waterfall. In some new variations of the modern Waterfall model, little overlapping and going back to preceding step is allowed, but the original model did not accept this kind of behavior.

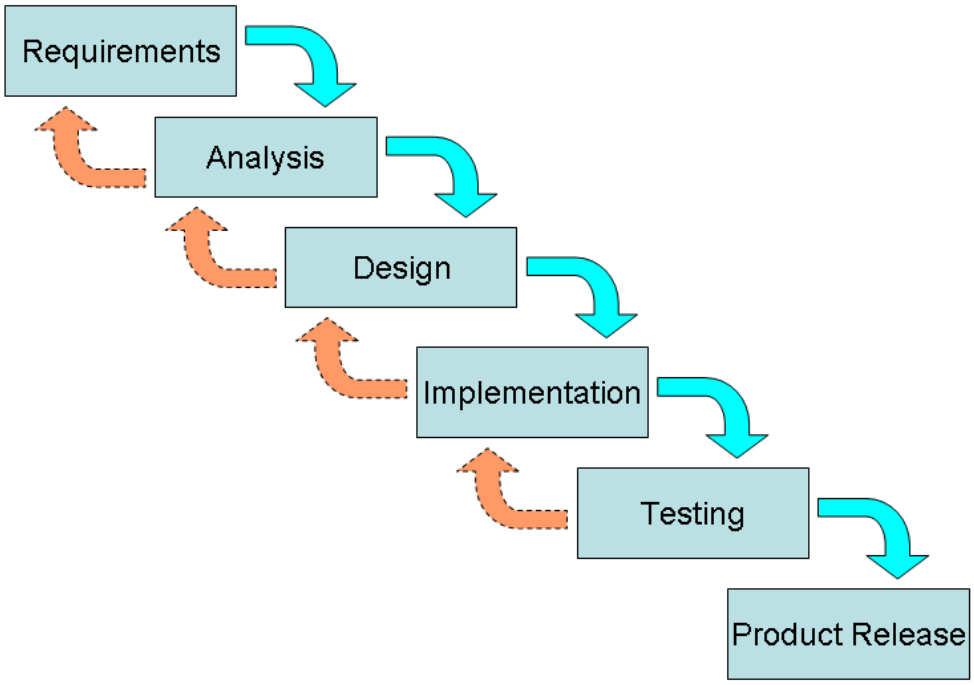

Figure 9: Waterfall development model

The discrete nature of this model makes it easy to follow. Every team in the project knows exactly what to do in each step. If everything is well defined, the specified deadlines are easy to accomplish. From a testing point of view this is a huge improvement compared to the two earlier models. All requirements are well defined and testers just have to follow specification when making test cases and when analyzing results. If the project follows the original model, finding a big defect from requirements it means that current release cannot be published and a new project have to be started to get the defect fixed. Also, the big downside is that other big changes after requirements are not allowed or at least welcomed, because it means a need to start from the point where the change is needed and do all proceeding steps again. In most cases the big changes are shifted to the next release. [4, 20, 22]

Spiral Development Model

Spiral model is an iterative model developed by Barry Boehm in 1986. The model combines elements from all three preceding models and adds iterative nature to the development. It means that there is no need to define and specify all requirements at once or to implement or test in one phase. The model suggests six steps iteration where smaller parts of the project are done in each iteration phase, until the product is ready to be released. These steps and spiral nature of the model is presented in *figure 10***.**

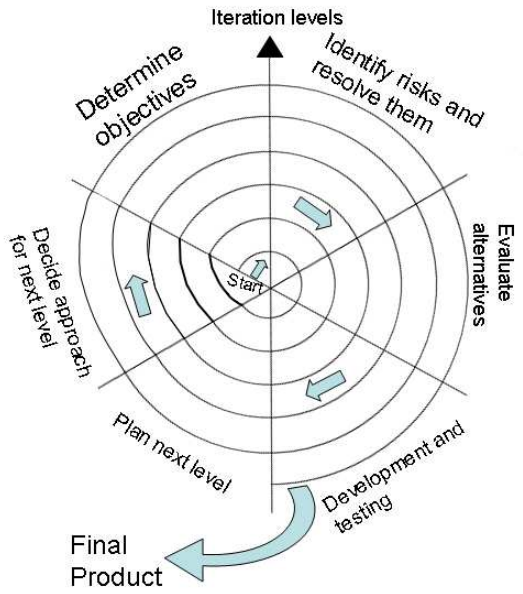

Figure 10: Spiral development model

Six steps are:

1. Determine objectives
2. Identify risks and resolve them
3. Evaluate alternatives
4. Development and testing
5. Plan next level
6. Decide approach for next level

The project should repeat these steps until the final product is ready to be released. From the testing point of view this is an easy task, because the tester will have specifications to follow and the possibility to change requirements for the next iteration level, if defects are found. [4, 20, 31]

5.2. Agile development models

Agile development models are the new approach in development of software. These models have gained place from traditional models, because agile models offer one major advantage against traditional models, the changes are welcomed during development cycles. Agile development models lean on four concepts presented in the Agile manifesto [32]:

"*We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*

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

That is, while there is value in the items on the right, we value the items on the left more."

Beside this manifesto agile development has twelve principles that every project using agile methods should follow. In those principles, the main reflection is developing working software, while keeping customer satisfaction in mind, with trust and honoring everybody involved in the development at the same time. Also, one of the key principles is to be able to deliver working software frequently, with shorter timescales. These concepts have been the key drive for the successful entrance of agile development model. [20, 22, 32]

In the projects point of view, agile development model offers solution in challenges that are out in modern software business like; time to market has been decreasing tremendously, minimize waste or unnecessary work also known as features that nobody will use, fast and easy correction of defects, requirements of products are changing more rapidly than earlier, development velocity in every area is increasing, and the life cycle of technologies is shorter nowadays. [22, 30]

Agile development models are iterative and incremental models where in each cycle there is the intention of doing only a small portion of the whole product. The difference between agile and waterfall development model against time and the iterative nature of agile development models can be observed in *figure 11*. In each cycle, every phase has to be completed to be ready for the next iteration round. If the cycle is unfinished at the end of the time window, it will be considered ongoing and the team will try to finish it before taking any other tasks. When this happens it means losing a little bit reputation as good and respected team in project. This way teams try to take only portions of work that they can for certain accomplish during the development cycle. Steps or phases needed to make this small proportion differ according to the used agile development model.

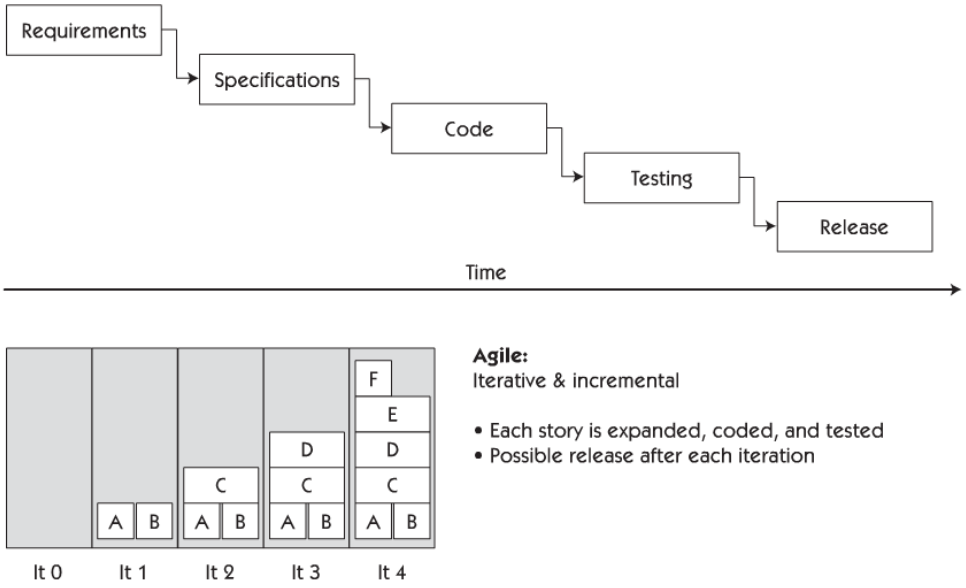

Figure 11: Agile versus Waterfall development model [22]

The two most known and popular agile development methods are Scrum and extreme programming (XP). Both methods tend to develop software in short iterative cycles and have different steps and tasks scheduled during these cycles. For example, Scrum calls development cycles as Sprints and in XP those are called iterations. Sprints and iterations have both fixed time window and sprint is recommended to last 2-4 weeks and iteration 1-3 weeks. In the beginning of each sprint there is a planning

meeting where user stories are selected, which represents tasks for the team, the same naming is also used XP method. Each day the team has daily scrums where yesterday's achievements are told as well as the tasks to be done today and the possible obstacles to achieve today's goal. In XP content is pretty much the same, but the name of meeting is the daily stand up meeting. At the end of sprint there are two meetings demo and retrospective in scrum method. The demo meeting is held to demonstrate new completed user stories and those are implemented to other developer teams and customer called product owner. Retrospective meeting is held to look at development ways of working used during sprint and how those could be improved. In XP, similar activities are held internally after the end of every iteration. There are not any official meetings. [33, 20, 22]

Testing in the agile development model is organized in a different way from the traditional development models where testing comes almost always as the last phase. In agile models testing is the key to make user stories to pass, without acceptance testing passing user stories are not ever accepted and ending. Many projects that have selected agile models also use test driven-development (TDD) and acceptance test driven-development methods as their way of working. In TDD workflow starts similar to the traditional models with planning and design, but as second phase or step is implementation of test cases that are expected to be passed when the corresponding part of code is ready. In TDD those test cases are at unit or module testing level. Then as third step the actual implementation and programming is started with the only requirement to make those test cases pass. When test cases pass, the implementation can be considered to be ready. The acceptance test driven development (ATDD) has the similar principle. The only difference is that test cases are done at functional testing level. The idea is to continuously execute those readymade tests to see when the implementation is ready. In a situation where the developer would think that implementation should be ready, but test are still in failed state, then those test cases and code have to be reviewed more carefully to see which ones are done incorrectly and make the change accordingly. These methods can be considered to emphasize testing, which is quite opposite to the traditional models. Advantages of making test cases first is to minimize waste implementation, detailed

specification in form of test cases, quick feedback loop and team´s concentration only in valuable things. These issues make development teams more aware about business and customer demands, which usually lead to better quality and increase velocity on development. [4, 22, 30, 33, 34]

5.3. Distributed Software Development

Distributed software development (DSD) has become a common practice for modern software companies around the world. In the literature DSD is also known and referred in some articles as global software development (GSD). The distinction between these two concepts is normally thin and vague; hence in this thesis those two concepts are considered and used as interchangeable. Definition of DSD is a development project that is divided between multiple working sites and locations, meaning that developers cannot work or meet face to face daily and to enable that developers have to travel [29].

There are many reasons for taking DSD into using it, such as the possibility to practice time zone independent 24 hour development, reduce cost by outsourcing part of development in low cost countries with access and obtain the best well-educated workforce etc. All this has been enabled by maturation of the technical infrastructure around the world. DSD is suffering from the same problems than single-site development such as lack of communication, inadequate or insufficient definition of requirements and specifications, cost and schedule problems and deterioration of quality. In DSD these problems are even emphasized and few more challenges can be discovered. If different sites are located in different time-zones with long distance from main development site there may exist cultural differences between sites. These challenges need special attention in making meeting arrangements and the development schedules, defining concepts which can be new and different for some working sites and making sure of sufficient and redundant network connections. Studies show that using DSD can take about 2.5 times more

effort to complete some tasks than one location based development, because of communication and coordination related challenges. [29, 30]

DAD is a special case of DSD. In the agile development model one basic principle that manifesto emphasizes is "*individuals and interaction over processes and tools*" [32]. Hence requirements can change rapidly since it demands for efficient and frequent communication between sites. Without this kind of communication there is the possibility that there are issues with lower awareness and poor coordination. This will lead to losing benefits of distributed development and can cause a negative impact on other sites. To avoid these possible problems there are recommendations to use various tools and means such as instant messaging software, videoconferencing, and desktop sharing programs [22]. Altogether, the main recommendation is to enable direct communication between developers to help avoid problems [30]. Other studies suggest that presence of customer or customer proxy is a key factor in keeping specification requirements and projects well coordinated [29]. Customer proxy is a person or team who acts as representative of actual customer and can make development decisions based on customer demands [29, 30].

Distributed development model impact on testing is similar to the distribution effect on development. Testing faces same challenges, requirements and possible problems can be avoided with same methods. Also mocked interface and modules used in unit test in different sites are causing a big challenge and increase in defects on integration testing. Most of the problems occurred when mocked module had outdated interfaces or message parameters. Few good practices and recommendations for testing are to keep test data, including test cases, test tools, configurations etc., available for all development personnel in all sites. Software under test should be at same baseline in all sites, before the testing is started. This enables testing to be done in the same manner and with the same configurations in all sites. Changes on testing tools have to be synchronized and informed clearly throughout all sites. [29, 30, 35]

5.4. Continuous integration process

Continuous integration process is a wide concept in which modular software is developed and tested in a continuous way, piece by piece. Martin Fowler describes in his article about CI, *"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible"* [36]. This is maybe one point of view where only unit and module level testing is considered to be part of continuous integration. *Figure 12* is presenting this kind of basic and traditional continuous integration workflow conception. In agile development model CI is especially important, because it enables having quick feedback on little changes which happen often and is one of the key building blocks of agile development. [22, 37, 38]

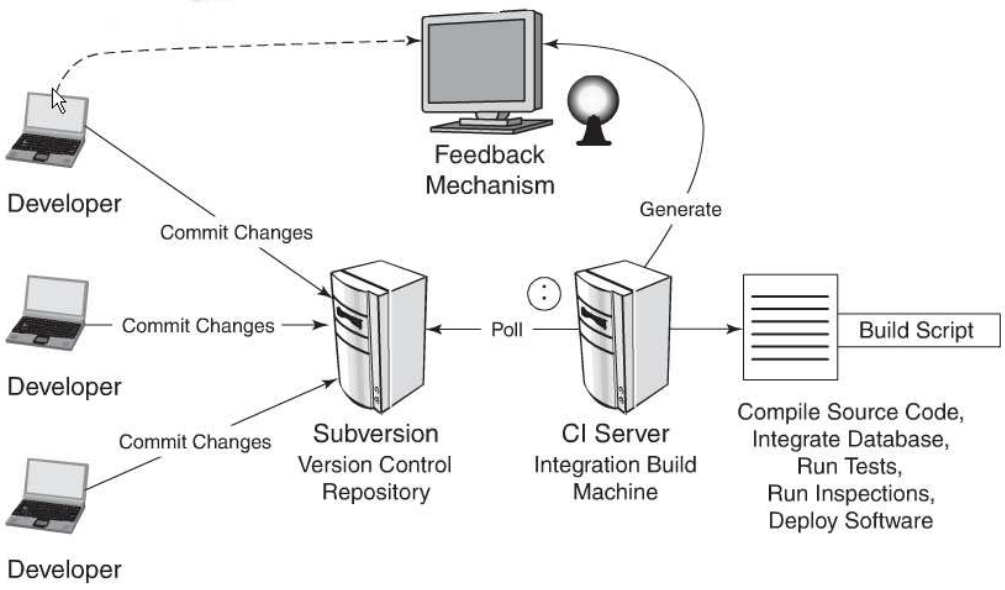

Figure 12: Components of CI system [38]

The most simplified CI process might not include anything else than automatic compiling of source code. Many consider that CI process without automated testing is not real CI, just an attempt to see if the code is compiling or not. One could

consider that the compiling process already tests the integrity of build, but without real testing the value of such process is just to verify the syntax of the programming language.

Continuous integration is mainly used to reduce development risks. By integration software changes multiple times a day and by doing it initiating automatic continuous integration process, defects are detected and can be fixed sooner. Usage of the automatic CI process reduces repetitive manual tasks and leads to the possibility of generating deployable software at any time. When integration is done many times a day it gives better visibility about the project's progress and status. For development teams it will enhance the confidence in the entire software product and increase motivation as a side-effect. Without automatic continuous integration process development usually suffers from late defect discovery, lack of project visibility and decreased quality of software. Disadvantages of CI can lead to lack of focus in overall architecture which eventually will lead to the architecture's degeneration. [37, 38, 39]

CI can be considered with larger scope than just integrating software blocks and executing unit or module test after successful compiling. CI systems have as pre-build phase's normal continuous integration code integration and unit or module tests. Then with some rule a complete build is compiled and tested with automated functional testing. The rule for making the complete build can be based on schedule or amount of integrated changes. Often complete CI systems can contain at least functional testing after complete build is compiled. This larger scope CI gives even more information about build's and software's overall health and increases confidence on the development's progress. [37, 38, 39]

6. Fully automated functional testing with Robot Framework

This chapter presents requirements, equipment environments and tools used to implement fully automated functional testing framework for MME network element. It will demonstrate the processes and methods used while achieving this fully automated test scheme goal. Also it will provide a general description of challenges met during this process and resolutions found for those. The chapter describes the actual decisions and actions taken during test automation scheme development and gives explanations to readers of why certain paths were chosen.

One of the agile development models key advantages is the ability to adapt for changes and increase development velocity. To achieve full capabilities of agile development one must use TDD and ATDD in testing. This requirement sets high demands for testing tools and usage of available testing environments. Tests have to be easily reproducible, modified and transferred to another environment, if needed. Also agile development sets new demands for regression testing, because in every sprint all previous functionality has to be verified again. This means full automation goal for every new functional test case test engineers will make. Project's goal for full automation sets a new demand for testing framework; all result report generation, log collection and test execution tasks in other words routine work has to be automated. With these demands fulfilled, test engineers are free to do more complex tasks which cannot be automated, such as designing and making new test cases. [34, 40]

Practical work Robot framework's testing library was at first done with agile development method as product development. For this task a Robot Test Automation team (Robot TA team) was founded. Later on it turned more like a supporting task which was not feasible to be done in agile sprints. Product development was still using agile; hence Robot TA team had to be aware of requirements set to testing

framework couple sprints beforehand. Further development of Robot test library in other hands was still carried on with agile development method. [25]

6.1. Test automation with Robot Framework

Robot Framework (RF) is an open source Python-based test automation framework for acceptance level testing based on generic keywords. Test cases are done by using tabular syntax in hyper text markup language (HTML) or in tab separated values (TSV) [41] files. RF comes with standard testing library set which includes commonly used functionality of testing methods. New libraries can be implemented either with Python or Java. [25, 42]

RF high level architecture can be seen in *figure 13* and it consists of four main levels: Test data, RF core, test library / test tools and SUT. Test data consists of test case and resource files. Resource files can be a collection of higher level user keywords, or variables. The RF engine is Python programming language based program which will interpret those user keywords written in test data and use test library keywords to execute commands in SUT. [25]

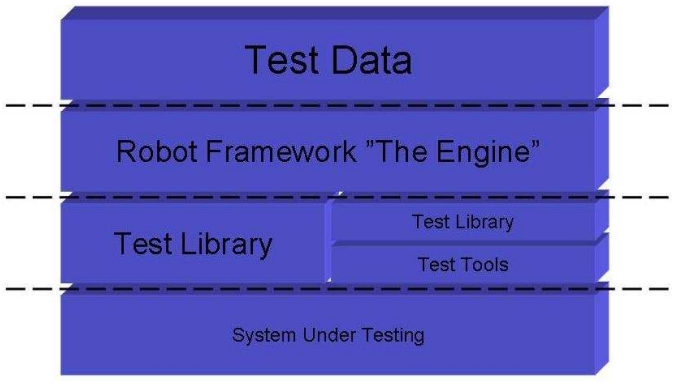

Figure 13: Robot Framework architecture

In RF there are two kinds of keywords: library keywords and user keywords. Library keywords are written in Test libraries with Python or Java programming language and those interact with lower level test tools or directly with SUT. So methods or functions in Python or Java are called library keywords in RF. User keywords are higher level keywords that combine those lower level keywords written in test library. Sometimes user and library keyword can be directly mapped without combining multiple lower level library keywords as user keyword. [25]

One of the main features of Robot Framework is tabular syntax test case and resource files. If the test case file contains multiple test cases it is actually referred as test suite. Also, the directory which contains multiple test case files is referred as test suite, and a use of hierarchical directory structure also divides RF suites in hierarchical structure. This structure follows the rules of Python hierarchy in variables and keywords. In those rules a child suite will inherit values and methods of a parent suite which can be overridden in the child suite. Also the rule of the nearest implicit name match is valid. These rules also apply to resource files. Only exceptions for these rules are globally defined variables which cannot be overridden. [25]

Test case file can contain four sections: Settings, Variables, Test Case and Keywords. The settings section is used to import testing libraries, resource and variable files, and also general metadata as a suite level documentation, common tags, setup and teardown for test cases or suite can be set in settings section. This metadata can be overridden in test case section, if the test case specific settings are needed. [25]

In the variables section there are different variables as integers, strings, lists and dictionaries for common suite level usage, that can be introduced. The test case section is the place where the actual logic resides and it is done by using a combination of available keywords. The last section of test case file is keywords where the higher level functionality is made by combining lower level keywords such as library and resource keywords to more convenient and practical form. Also

loose functionality as loops and condition statements can be made to make actual test case section more readable and compact. An example of test case file can be seen in *Figure 14.*

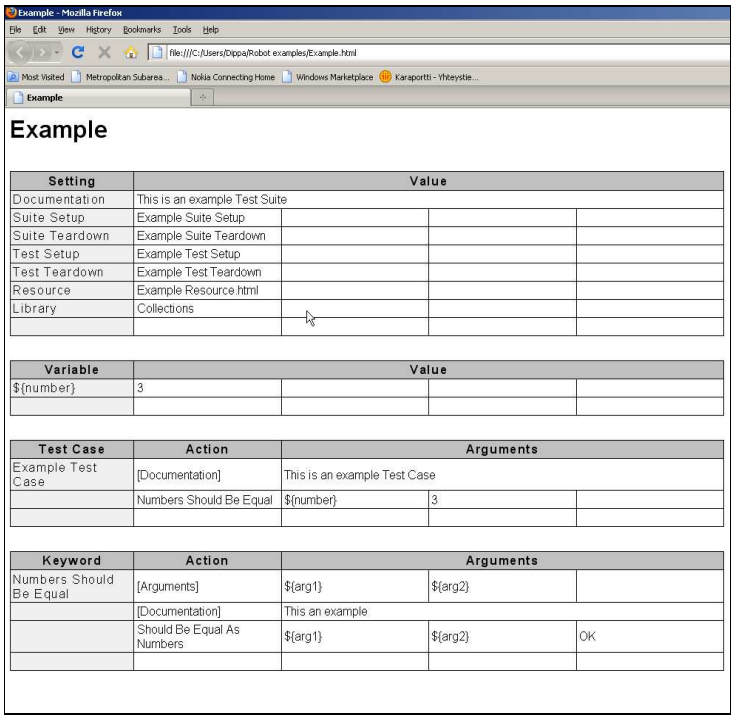

Figure 14: An example Test Case file

Test cases can be executed based on test case file or name, directory and tags. Tags are an option to classifying test cases by using free text. Reports and logs are also showing statistics of different tags and reports that can be set to even organize test cases based on tags. In test case execution test cases can be excluded or included based on tags or set as critical or non-critical for results. Robot Framework uses command line interface to execute test cases and follow progress outputs of execution, which are not very informative as you can see in *figure 15.* [25]

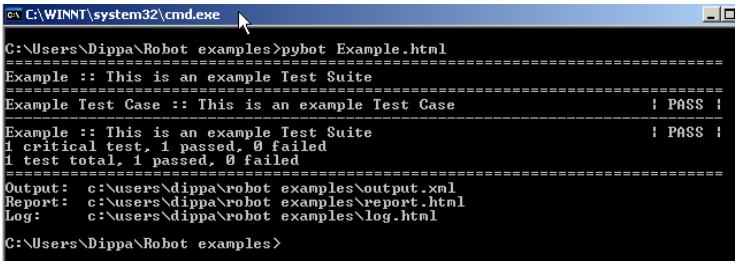

Figure 15: Robot Framework CLI

There is the possibility to view more specific information about execution through tailing debug file. As for the test automation point of view this lack of interactive view does not cause any inconvenience, but for the development of a new test case it has been found as a restraining factor. Outputs of execution are XML based output file for integration with other possible continuous integration systems and HTML based report and log files for a human readable format, which can be seen in *figure 16*. All test data files are in textual format and can be stored and maintained easily with a version control system. This makes it possible to store test data, together with products source code. [25, 43]

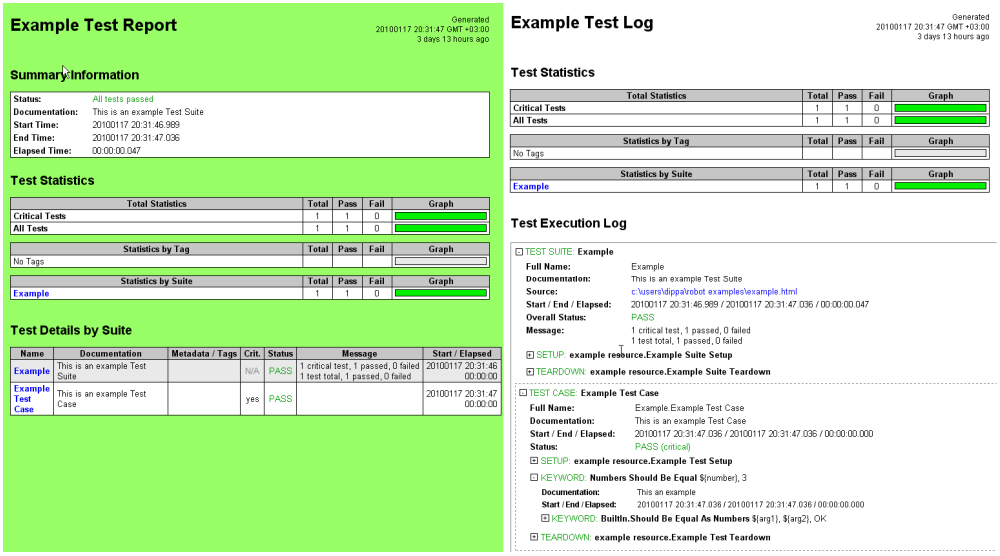

Figure 16:Example report and log file

Robot Framework is a Python based tool which is interpreted language and does not need to be compiled before execution. RF offers simple library application

programming interface (API) which can be used to extend, enhance or make new Test Libraries. Test Libraries can be done with Python or Java, which are both interpreted programming languages. This feature makes it easy to update and enhance both RF and Test Data. Interpreted language is also platform independent, only an interpreter is needed to install. There are already many commonly used Test Libraries available at project homepage, for example Selenium for web application testing, and libraries to use SSH, Telnet etc. [25, 43]

6.2. Development of Robot Framework testing library

Development of Robot test library was done with the agile development model by Robot TA team. It was started at the same time as products development. The MME's product development started with tasks related to general software architecture design, framework studies and planning. Thus at first only module and unit testing was the only feasible testing method for code. There was no need for E2E testing framework right away.

Our team started by choosing suitable tools, getting familiar, and training to use those tools. Team started to develop our own Robot test library for the MME element. Right from the beginning, our goal was to design and implement the library in a way that it could support fully automated testing. Also guidelines and instructions on how to use our testing framework should be made in ATDD style. Rethinking of test automation was our team's continuous task. At first, how we could make one test case automated and make it independent from other test cases. Proceeding with design task; how could we execute test cases in test suites. [25, 40]

6.2.1. Getting info, formal training and self learning

The project started with gathering information from RF's project's webpage and the user guide. Formal training on Python programming and Robot Framework for early stage developers was arranged. Robot Framework training was held and given by RF developers. The direct formal training was needed to give our team a kick-start. At

the same time, took place the introduction to agile ways of working, LTE network architecture concerning especially MME network element and other tools for example: subversion-control (SVN), LTE network emulators and Pydev an Eclipse plug-in which is integrated development environment for Python. [42, 44, 45, 46]

After formal training there was a couple of week's time to get hands on experience with new tools and the environment. During this time, a new RF test automation team was founded, including 5 full-time members and 2 part-time more experienced testing specialists. This hands-on training period was used to get more familiar with tools and make a couple different kinds of architectural models for our future testing framework. At the end of this period, models and ideas were gathered and merged as one architectural structure for our future development.

6.2.2. Beginning of Robot Framework Tester Development

In the second phase right after the formal training for RF test automation team, the actual development, designing and implementation started with the help of RF developers. One of the first steps was gathering knowledge on how the test library should be organized and done effectively. Mindset was to make this library and testing framework suitable for fully automated testing, so no hard coded values or manual process steps would be allowed. This work was scheduled to be done in agile mode and using Scrum method in two weeks sprints. [47]

Development started from control library for LTE network element emulators which act as real network elements in testing environment. Design and development started from zero and the first step was enabling connections to emulator PC hardware and SUT. The MME network element was still in an early development stage and there was no possibility to take mature enough release which could be used for end to end (E2E) testing purposes. User interface was planned to be similar to the one used in previous generations SGSN hardware and so SGSN was decided to be used as first test environment for our library. LTE emulators acted in similar ways to the previous

3G or 2G emulator releases, which eased our kick start. This meant that according to the testing library development point of view, the basic knowledge and all needed resources were available.

In this early stage of development, the decision was made to divide test data and development into two separate parts. In the first part, the *PythonSource* test library included MME specific RF Test Library made with Python programming language and supporting tools. Other part *Testers* designed to include all other RF related files, such as html based test case inside *FeatureX* folders, resource files in *Resources* and all needed environment specific configuration files in *EnvironmentLibrary* folder. F*igure 17* shows this early stage tester structure. The first design of the test library contained only EmuLib for emulator control and platform specific DxMml part for connectivity and basic commands to SUT. For keeping files and folder harmonic and solid, it was decided store it under SVN. SVN made it easy to track and revert any changes, if needed. SVN was a very feasible solution, because all files were used in textual format [46].

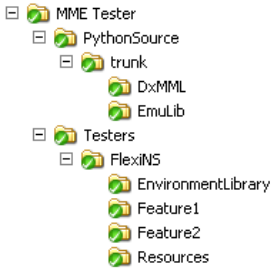

Figure 17: MME Tester structure, first design

The full automation target was one of the team's main goals since the beginning. Another requirement to reach the full automation target was the ability to execute all tests in any free environment. All testing environments were corresponding to each other in testing software- and hardware-wise, only differences are in configuration like IP-addresses, identification numbers etc. This resemblance made full automation more feasible. In RF test cases, it was decided to implement it through parameterization of all needed variables.

All constant variables should be written in resource or test case files and environment specific are read and parsed from environment specific configuration files in *EnvironmentLibrary.* Those files are written as text files with environment name plus underscore plus config like *IPC-ATCA-1_config* and as postfix *.py* which will make them Python code files. This makes the usage of these files easy inside the RF testing library, because those can be imported directly in library source code files or into RF test cases. Then variables are valid to use without any parsing etc., only requirement is that configuration files are written according to Python syntax. This allows the usage of all necessary environment specific variables and parameterization of test cases. There are only two mandatory parameters which have to be given as global variables in RF execution command. The first one is *TESTENV,* which will be the name of the environment where the execution takes place e.g. *IPC-ATCA-1*. The second one is *WID,* which is the short form for worker identification. *WID* is used to map user specific information, for example test tools paths which the user is using and those paths are defined in environment specific configuration files. For automation and CI there is a special *WID* which launches other activities, like automatic test data update from SVN etc. This way, all configurations, settings and variables can be easily mapped for certain environments and all tests are truly independent and can be executed in every environment.

After design and implementation of the first version of the RF test library, a test phase of testing framework took place. Tests were more proof of concept style than actual functional or defect finding tests. All tests were only empirical and based on showing demo of functionality and capabilities of Robot Framework with the assumed product. No preliminary testing plan was written, only preliminary requirements were available beforehand. Demo was successful and RF was accepted as the main testing framework for MME network element. After demo RF test automation team was dispersed and reorganized for future challenges.

7. Integration testing and tool development

This chapter describes actions and tasks needed during and after shifting from pure development to production phase. After designing, the implementation and proof of the concept demo of RF testing library's first version, started tasks for support, training users and competence transfer. The MME network element reached enough maturity to start E2E testing. First real E2E tests with real element put RF testing framework to production which caused a lot of support and error correction as maintenance tasks for our team. At the same time, demand for new keywords and functionality for testing library increased greatly. Real E2E testing and feedback from users, lead to the redesigning of the testing library structure.

First E2E testing was done with NSN proprietary LTE network element emulator software. Emulator software uses fuzzy methods when accepting received messages and does not inspect every value inside the message parameters to be correct. This makes it more feasible in preliminary testing, where only certain parameters and the total message size is evaluated. Second phase and more sophisticate testing was done with testing and test control notation version 3 (TTCN3) tester. TTCN3 tester's development and compiler was acquired from and done by third parties. This was necessary to guarantee unbiased and independent testing results. [48]

Acquirement of a new component like TTCN3 tester for E2E testing also required support from the test automation framework. Design principles of framework were also modified to support different testers in the future without massive redesign or implementation. This work was carried out by the Robot test automation team along with competence transfer and training for Robot framework users. The first real testing experiences with Robot framework led to some enhancement requests for the framework. This initiated a redesign process to one part of the Robot framework structure, to make it more suitable for test development and test data handling.

7.1. TTCN3 testing language and tester

Testing and Test Control Notation Version 3 is an internationally standardized language for writing and controlling tests. The language has been developed and is still maintained under the Methods for Testing and Specification Technical Committee (TC-MTS) at ETSI. This group consists of the leading experts from the testing community organizations and major industrial members. TTCN3 is based on Tree and Tabular Combined Notation version 2 (TTCN2) which is the preceding language of TTCN, developed and maintained by the same group. The language changed its name, because version 3 does no longer use tabular format and is more like conventional programming language. TTCN has been over 15 years the standardized testing language and is widely used by the software industry. Version 3 had its first standards in year 2000 and has been stable ever since. [48]

TTCN3 standards are accepted and followed by the testing tool industry. The language can be used to specify test cases and with them verify standardized or proprietary solutions. It has been already used to make tests for complex and very large industrial systems for example in telecommunication for 3G systems. TTCN3 testing language looks and feels like a conventional programming language and it has well defined and standardized syntax. It has been designed for testing purposes and has a couple ready-made embedded special testing features like timers, verdict and native list types, subtyping etc. TTCN3 language also supports its test components usage as emulated interfaces, which reduce test environment complexity and maintenance load. This enables a completely automated testing environment and test execution. [48]

TTCN3 language itself is not executable and always requires a compiler or interpreter to be operational. There are many commercial off-the-shelf tools and test systems already available and open source tools currently under development. Adoption of the TTCN3 language can be quite straightforward and easy with these off-the-shelf tools. [48]

TTCN3 tester or actually compiler which is used in testing of MME was developed by a company called Telelogic, now acquired by IBM and added to IBM's Rational Software family [49]. Besides the compiler there is the need for variable, message, codec, function and procedure definitions for the tester to function. This work, which is actually tester development and enhancement, is carried out by a third party for us. To make tester according to 3GPP standard specifications is a very demanding work and has to be done by an outside contractor to preserve an independent, unbiased and objective point of view. Also, for customers it is a sign of unbiased testing when the test tool is built by an independent third party.

In RF there was need to build an interface for controlling the TTCN3 tester. At first it only needed to start a few sub processes for the main TTCN3 tester process and collect logs and analyze the verdict from standard out. And besides this, there would be monitoring and log collecting tasks from SUT. TTCN3 tester tool carried out the actual testing via interfaces of SUT as shown in this setting in *figure 19.*

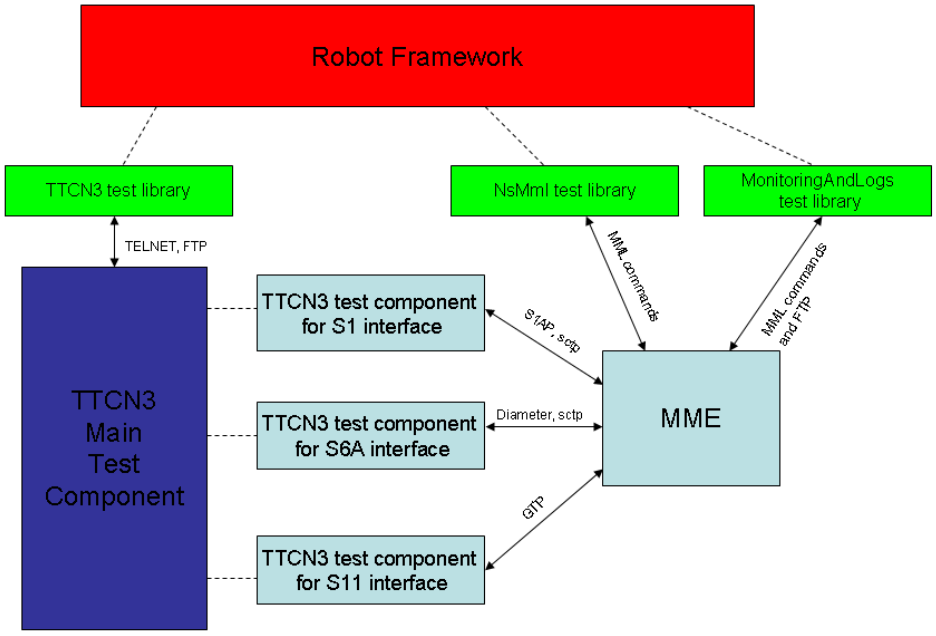

Figure 18: Robot Framework and TTCN3 combination

Another main feature was also the parameterization of variables used in TTCN3 test cases. Without this parameterization full automation cannot be achieved. Variables are stored in the environment specific parameter file in TTCN3 testers' installation path, which TTCN3 uses when executing test cases. Information on which parameter file TTCN3 should be used is given as parameter in the startup script. In MME development this file is named with the same environment name prefix which is used in *TESTENV* parameter in RF and postfix *.par* e.g. *IPC-ATCA-1.par*. To make sure the same values of variables are used in RF and TTCN3 tester, in RF a feature has been built to read and parse variables from environment specific TTCN3 parameter file at setup phase of each test execution cycle. This way variable values are exactly the same in RF and TTCN3 tester. Parameterization also supports the goal of a fully automated testing environment, because in test case there isn't any hard coded variable values and all variables are taken either from the environment parameter file or used default values defined in TTCN3 tester, which are not environment specific.

Later on there came up a need to make check ups and interrogate information from SUT during the test execution. This kind of feature helps to verify MML user interface commands and outputs inside SUT, but the most important feature is giving commands through MML or service terminal which will initiate a procedure. A good example of this procedure, that can be only triggered from inside the SUT, deleting the existing and active subscriber from MME which should lead to gracefully teardown all subscribers connections. This kind of feature is necessary for example when the element is going under maintenance break and its load has to be moved to another element. To achieve network detachment, it has to be done inside the element and the functionality cannot be triggered outside the SUT. To allow this, a synchronization and signaling channel between the TTCN3 tester and RF was designed and implemented.

Synchronization needed implementation both in RF and TTCN3 tester. In TTCN3 tester there was already the generic implementation for external synchronization available, but it needed some modifications to be usable and compatible with RF. The TTCN3 tester was already doing internal synchronization between testing

components for helping out timing issues and improve visibility and debugging features. The basic idea of synchronization between test components is quite simple. In TTCN3 test case creation all test components which are called Parallel Test Components (PTC) are defined in the Main Test Component (MTC) also synchronization points are named and number of synchronization parties are written here. When test case execution starts MTC starts to wait for PTC's synchronization messages and when PTC reaches the synchronization point in its own test execution flow, it sends the synchronization message with verdict to MTC and starts waiting for the reply from MTC. When MTC has got all synchronization messages from PTCs with a successful verdict, it will send a "go ahead" signal to all synchronization parties and they will carry on the test case execution. This synchronization procedure will loop until all synchronization points are completed. If the PTC synchronization message verdict is not successful, MTC will immediately send a stop signal to all PTC's, because there is no point to continue testing after some of the PTC's have failed. Graphical presentation of this synchronization flow can be seen in *figure 20.*

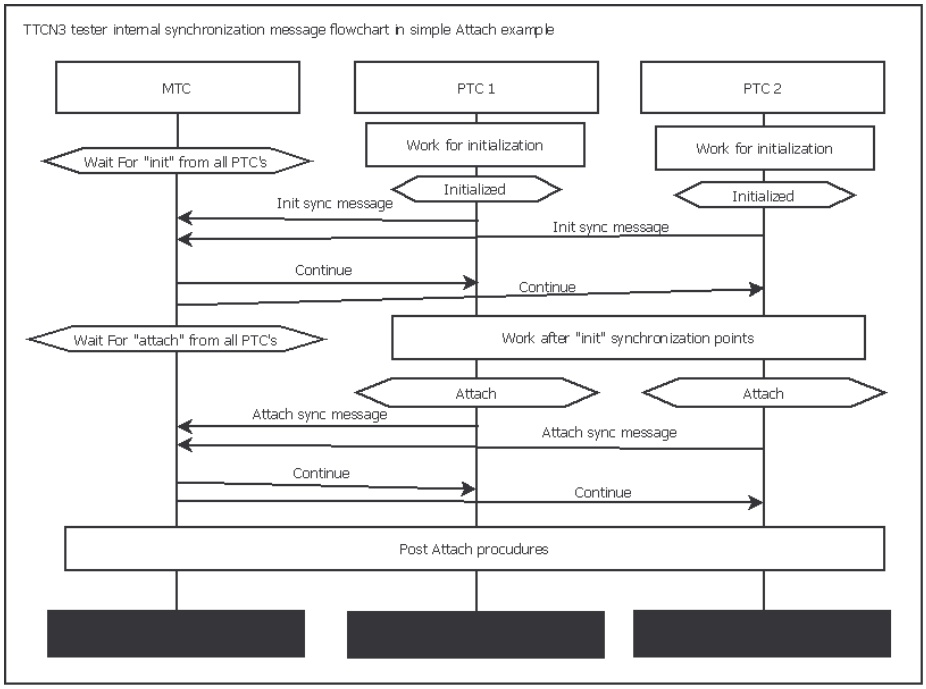

Figure 19: TTCN3 tester internal synchonization example

The synchronization between RF and TTCN3 tester works quite similar to the TTCN3 tester's internal synchronization with few exceptions. The first exception is the initialization of a signaling channel between the TTCN3 tester and RF. The signaling channel uses server - client model where the TTCN3 tester acts as a server side and RF is a client. Communication is done by using telnet protocol [50] in port 55555, because normal 23 port is already used by OS telnet server service and it would not be feasible to use reserved ports [51] to implement proprietary solution. The RF is monitoring TTCN3 tester's standard output stream to catch certain indication string, that initialization of synchronization signaling channel server is done and RF can connect to port 55555. Then a simple handshake is carried out and the signaling channel is ready to be used.

The second exception is communication flow between PTCs and MTC in a case of external synchronization point in use. In the TTCN3 tester only one PTC can be using this external synchronization with RF, because current simple implementation of this protocol cannot tell the difference between different PTCs. As for the rest of PTCs these external synchronization points are invisible and look like internal synchronization points. When the PTC reach external synchronization point with successful internal verdict, meaning all procedures before synchronization have been successful, first it will communicate this reaching of synchronization point to the external interface which is external synchronization signaling channel server process. The server process will deliver this message to RF, which now can act and do some check up, interrogate or command functions. After RF has done functions defined in RF test case it will return the verdict of those actions with a simple *PASS* or *FAIL* message. Then PTC will deliver this verdict to MTC, which will then give the verdict to continue or teardown to all PTCs. This helps the TTCN3 tester and RF to terminate the test case execution gracefully. A part of the successful flow of this external synchronization can be seen in *figure 21*.

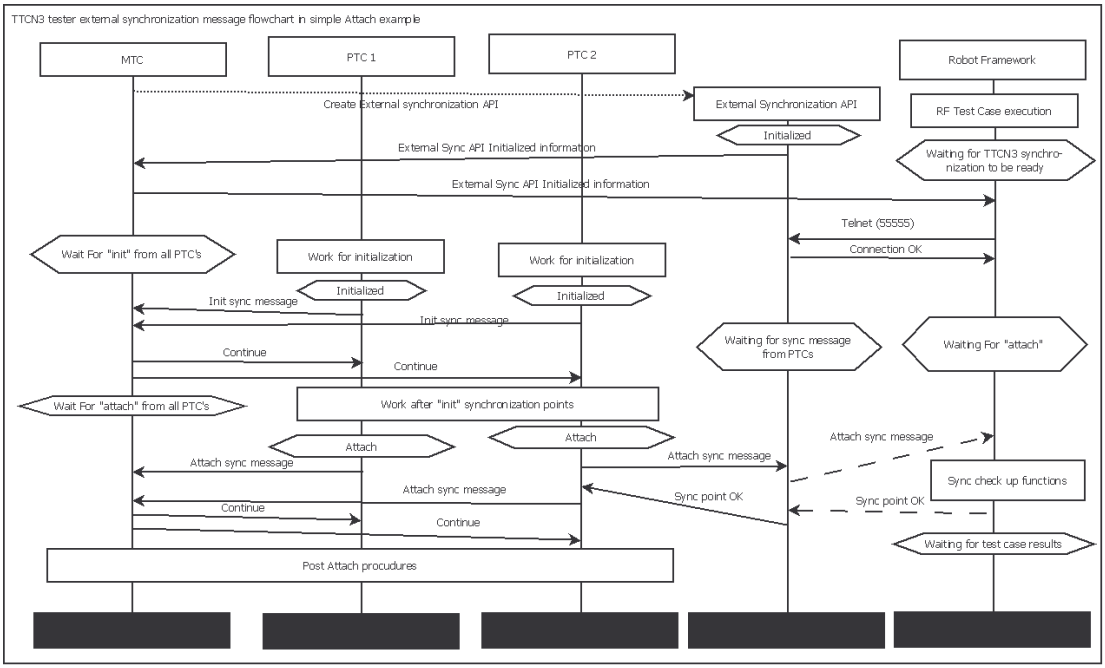

Figure 20: TTCN3 tester and RF external sychronization example

For unsuccessful execution of other PTCs test case during or even before external synchronization point is reached will lead premature teardown of TTCN3 tester and with it lose synchronization signaling channel or at least never arriving of a synchronization message or RF can crash or terminate without sending reply. For these kinds of events default timeouts have been declared for both TTCN3 and RF. These timeouts are made to be changeable.

The need to communicate verdict of synchronization point back to TTCN3 tester from RF side led to the implementation of a special TTCN3 analyze library. Normally RF will fail and stop test case execution when some keyword fails, but in check up, interrogate or command keywords during TTCN3 external synchronization point this is not feasible, because then the TTCN3 tester would wait for resolution until timeout would occur. TTCN3 timeout has been set by default, being quite long, because changing it is not so easy and flexible. With TTCN3 analyze library the RF will never fail on error, but store this result of interrogations, checkups or command results and first return negative verdict to TTCN3 tester and wait for the tester to finish. The similar process flow is also with error free executions with the exception

that execution continues until all synchronization points are completed. At the end, the result is analyzed according to the TTCN3 output and possible errors are printed out to log.

7.2. Start of integration and end to end functionality testing

Nowadays software of complex systems is usually built in a modular structure. In modular structures program blocks can be developed separately and tested only with unit or module testing in which other blocks can be mocked. And even though module testing would be done with real program blocks, there is seldom enough module test coverage to cover all possible situations with external interfaces of program blocks and their signals. When different separately developed program blocks are tested together it is called integration testing, as seen in an illustrative *figure 18.* [37]

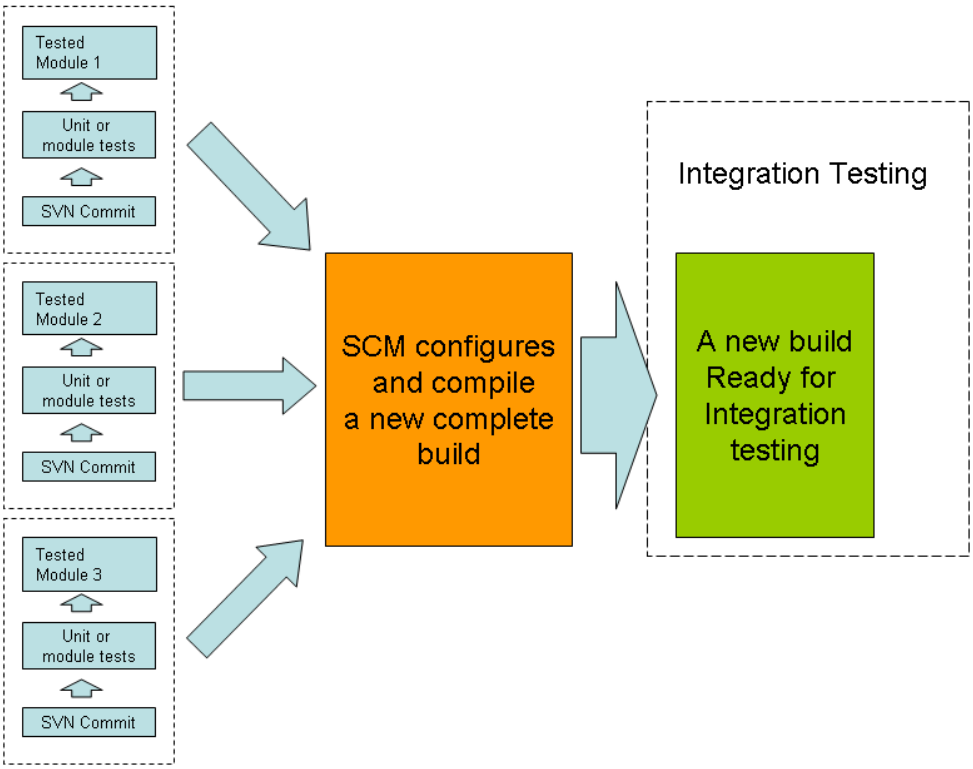

Figure 21: Modules combined for integration testing

The MME network element application software is developed in multisite environments, where sites are geographically located in different places and even in different time zones. This makes integration testing even more challenging and important, because real time change distribution is not possible. This means that not all program blocks or code changes are distributed in all environments simultaneously. Only a few times a week, a new complete build is installed in every site to test environments. To keep the external interface intact, synchronized and compatible with other program blocks, development teams have common variable definition storage, so called public definition environment (PDE), where each message between program blocks and their parameters are defined. So each site is required to do only unit or module testing and after it is passed, the program blocks are compiled again in the main site, where the complete MME software is built and tested for the first time in the integration testing. Besides the application software in complete MME element software there is a platform, in other words, an operating system which is also updated on certain schedule. Although the platform's operation should be invisible to the application, except for some APIs that the platform is supporting, there are always issues or defects that do not come up in the plain platform testing and are only found through the application testing. Those faults and compatibility are also tested for the first time in the integration testing and possible problems are reported to the platform development. [30, 37]

In the beginning of the integration testing, the only tool to test E2E functionality was NSN proprietary tool, LTE network element emulator. As a normal emulator, it is a non-deterministic and stateless tool and does not take care of previous or following states subscriber or SUT. There were two main reasons for using the emulator as a preliminary testing tool. The first reason was the need for loose or fuzzy message content handling from a tester. This gives more tolerance for errors in the message content. The second reason was unsuitableness of deterministic TTCN3 tester which was more demanding of a message content verification tool. TTCN3 tester would also been possible to be configured and build to accept more inexact or fuzzy messages, but it would not have been feasible, because the LTE emulator was already capable to validate messages with enough precision. The decision was made

to use the LTE emulator as preliminary E2E testing tool in the beginning of the integration testing. TTCN3 tester was selected to be used in more strict and sophisticated E2E testing and it could be used in integration testing when the product was mature enough. Using LTE emulators as an early phase validation tool also released TTCN3 tester development resources to be used in more demanding tasks instead of maintenance work.

Test case development is always done as manual work. When using LTE emulators this means starting at least 5 or 6 different processes in a certain order. And after initialization you still need to give a few macro commands and analyze standard output of emulators to find out the result of test and determine if it did go as planned. For test case development purposes, this gives real time interaction and feedback on the test event. But reuse and repeatability are not so high, because the manual work timing is never the same and risk for human errors is apparent. RF gives fair advantage and ease in reuse or repeatability of test cases. It takes care of emulators' initialization processes and timing differences can be measured in milliseconds instead of seconds. Another great advantage compared to manual testing comes from collecting log and monitoring files after tests are complete. In manual testing you have to collect logs from those same 5 or 6 processes and besides that, SUT also offers various log and internal message monitoring files which help one to verify, analyze and debug test case events. In the RF testing library implementation all those simple and repetitive tasks are carried out automatically. This makes once tested and approved test case execution faster and easier reproduce.

7.3. Training and competence transfer for users

The task of adopting, acquiring and seeking knowledge of RF changed to sharing and giving competence transfer to future users of RF testing framework. This meant an attempt to convert existing information and tacit knowledge to training materials and events. Besides a new testing framework also the working method and product were completely new and continuously changing, so this made the challenge even bigger. New methods and tools always cause change resistance which lead to the decision to use hands on or learning by doing methods such as primary training methodology when passing knowledge and information to new users. Teams were using TTD in module testing and in entity testing ATDD was chosen to be the future method. Training and teaching ATDD method to users was not included in the first phase plan. [34, 40]

The first phase of the training and competence transfer was designed in three parts. First part was general information of RF, related tools and their advantages. The first part was targeted to all teams and persons working in MME development and for that reason it had to be a light and quick presentation of RF. The second part was a more detailed description of RF libraries, basic functionality and modifications made for MME's testing purposes. Also the recommended tester structure, the use of SVN and the recommended tools were introduced. The third part was most significant and important for future users and was arranged as a hands on and learning by doing. This way training and competence transfer could be modified according to each team's needs and preferences. A hands on workshop was held first for teams with all test engineers and continued with other similar sessions, if needed. After key users got their training the assumption and idea was that they will share it to other users and RF TA team would only give two first parts of training for new comers and give support for key users in difficult cases, if needed. At first the need for support was quite high as expected, but soon after key users absorbed more information and turned it into tacit knowledge, and requests for basic and simple problems vanished.

7.4. Testing framework structure redesign

RF testing framework is under continuous development and enhancements. Need for change comes from continuously changing SUT and testing tools. At the beginning the structure was considered only from a higher architectural perspective as only test library and test data separation practices. As test case amount and future usage model was coming clearer, the design needed some improvement for test data part. In *figure 22* this change is presented as it was in design phase. In test library *PythonSource* – folder one can see two folders *tags* and *trunk*, after a short experimental period of frozen versions of RF test library, also called tags, was decided to wind up. The need and schedule for enhancements and new features was too fast for periodic development and so the decision was to use trunk –branch as our production revision also. This decision required more focus on testing, since even every minor change has to be tested thoroughly before committing it to SVN. And other visible changes are Libdoc which is used to create automatic documentation of test library, MonitoringAndLogs library which contains log collecting and SUT monitoring functionality, NsMml which contains SUT specific testing functionality, NsUnitTest is experimental library to make unit tests for the test library and last Ttcn3library, which is used to control and interact with TTCN3 tester.

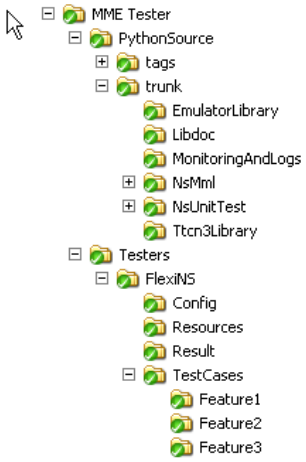

Figure 22: Redesing of MME tester

In the *Testers* side the change is more administrative than operational. Earlier environments configuration information files were stored under *EnvironmentLibrary*, in the new structure the name is more descriptive *Config*. The *Resource* folder has stayed the same, as the folder for RF common resource files. A completely new folder stored in SVN is the *Result* folder, the meaning of it is not to store results from every execution to SVN, but quite the opposite. It is only in SVN to clarify where the recommended place to store the execution results is. Because in RF you can specify the output folder as you like, but in examples this *Result* folder is used. To make sure no one will store execution results in the SVN, it is forbidden with administrative measures.

The last improvement is for actual test cases which are organized under the *TestCases* folder. Their recommendation is to use a naming style coming from agile development where first comes the epic name and subfolders beneath are named after the user story and id number. This will help tremendously in search and update tasks. This also gives the opportunity to execute all test cases easily with RF, by just giving root folder *TestCases* as an argument for test data location and adjust execution assembly and settings with tags and arguments. [25]

8. Automated Functional testing

This chapter describes and answers the following questions; Why automated functional testing is needed for the MME network element, what is needed to enable this automation and how is this automated testing is carried out with different testing sets? The amount of needed test cases to have a good or at least moderate test coverage is rapidly increasing in complex telecommunication systems.[3] To execute all or even some applicable test case sets for every build and version of software manually is very time consuming, monotonous and a wearisome task for test engineers. Nowadays it is becoming even less feasible due the continuously accelerating development schedule. There is no room for errors in the execution, logs have to be collected and reports have to be made from each execution run. And the precious work effort of test engineers is needed in making and analyzing those test cases, not in their execution. All these reasons speak up for the importance of fully automated test execution framework.

With the increasing amount of test cases and spare time of environments during nights, an experiment idea came to execute those test cases as one set. The idea was to test possible limits and robustness of RF implementation and the MME network element. This meant an experiment where for the first time a larger set of test cases would be executed in a consecutive manner. This started as an experiment held by the Robot test automation team, but after awhile the word of this experiment and its good results spread around and the decision to take this set as a part of the production tools was made. After making this an official production tool, it was named; nightly regression test set. Also smaller automated smoke test set was invented as a side product at the same time to make it quicker to test new builds sanity. Both of these experimental side products would help everybody in development to see sanity and functionality of our product from reports. Later on both of these experimental products were taken as one official meter of our products sanity and progress.

Although the mind set and goal is to make all test cases automated without need for human interaction, it is not always feasible or even possible. One has to always consider the ratio of benefits and resources that the automation will obtain and take. Of course, some very rare cases like sudden outage of some units cannot be always done with test tools and has to be done manually by taking the unit physically out from system. Also other test scenarios and exploratory testing which cannot be feasibly implemented with testing tools has to be considered. These tests need to be documented very carefully and use automation as much as possible to make them reproducible.

8.1. Smoke and Regression test sets

In testing there is always the basic question on how to get good enough test coverage in feasible time and how much testing is enough? For minor changes one test case can seem feasible, but if such change will affect base or root functionality and have a general impact, the code and testing have to be done for all sub functionalities. This kind of all over coverage testing in MME development and in software development is usually referred to as regression testing and is designed to cover, as much as feasible, all relevant test cases. Relevant test cases are determined by the team of test specialists led by the test architect and those tests should cover at least all features and functionalities, but leave out only duplicate testing, if time window limits test execution. In agile development there is also a need to get quick feedback on current builds status or sanity. This kind of test set will contain only basic test cases with basic features and in agile development it is usually referred as smoke test set. The smoke test set will give green light to further testing phases, such as regression. Main differences of these two sets are gathered in *table 2.*

Test Set	Regression	Smoke
Execution time	Long, from couple hours to days	Quick, less than 30 minutes
Main purpose	Test that new code has not broken the old implemented functionality	Test root functionality, check sanity of build
Execution schedule	Daily (night time)	Whenever new build is installed
Test set	All inclusive, includes also the smoke test set	Highly focused, well targeted for root functionality
Results significance	All smoke tests must pass, major percentage of other tests should pass	All tests must pass

Table 2: Main differences of Regression and Smoke sets

The regression testing is usually all inclusive and hence takes quite a long time to execute. Basic idea behind regression testing is not to find new defects or faults, but to try to indicate, if the new code will break the old implementation. Therefore, all test cases which are added to the regression set should have already working implementation and cases should have passed test at least when implementing and having demo of the code. Of course, it can be that those test cases are executed only in short runs or series during their development phase and hence, new faults are emerging and visible only during long consecutive execution of the whole regression set. Also the effects of previous events in systems under test can have affect the results of test cases. Therefore regression testing does not only test all functionalities of SUT, but also the stability and robustness of test cases as well. One of the main requirements for test cases is independency from other cases. This means good enough set up and teardown functionality of test suite and cases. This can be tested by executing test cases in random order each night. Random execution also makes sure, that SUT will be tested in various testing scenarios which can reveal new kinds of faults. In MME development regression is meant to be executed at least daily and

the target is to get results ready by the next morning, so teams can check how yesterday's changes have affected the build.

The smoke test set is more narrow, and highly focused than the regression test set and targeted to only the most important features and functionality. Its main purpose is to check builds state, if the root functionality is working like it should, or if it is broken somehow. If smoke tests will fail, it means no further steps in testing are feasible to do, before faults in smoke functionality are corrected. This means stating failing build as broken one. This is a very clear signal to development teams that there is an A-class problem. Focus of smoke test is usually in root functionality like in MME development the attach procedure, if attach is broken it means roughly that more than 95% of features and functionalities don't work at all in that build. Other smoke tests are selected from the same kind of areas where the root functionality is tested. The smoke FT test set should be quick to execute and give results in less than 30 minutes. Time is critical in smoke testing, because if a fault is detected there cannot be any commitments besides the code to correct the emerged fault. If smoke testing takes a long time, it will affect the teams' velocity to produce a new code. The smoke test set is also a good way to see how some minor change on the testing framework, tools etc. is affecting our testing environment.

Making different kinds of test sets is quite easy with RF by using tagging. Tags are free text and can be added to each test case separately or for suite. Tags can be also forced on, so each test case under the suite will get that tag. The test case selection can be done in execution command by including or excluding test cases based on the tags that those contain. For example shown *figure 23*, smoke test set can be selected

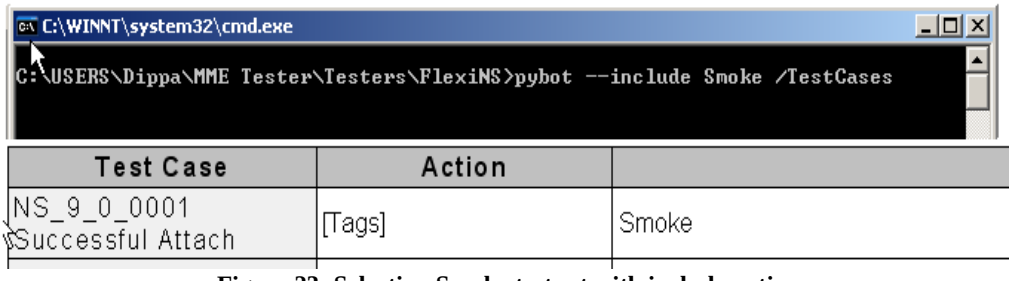

Figure 23: Selection Smoke test set with include option

by giving RF execution command with includes an option which will select then all test cases with “Smoke” tag on them. The method is also used to select the regression test set and exclude some long or problematic test cases. And using tags include and exclude option in RF you can easily influence the execution order by using multiple execution commands where inclusions and exclusions are in certain order and then combine the results with a supporting tool called rebot. Rebot is a robot framework’s supporting tool which can be used to adjust look-and-feel of RF reports and also combine different test execution results. In MME development tags are used to select the test set to smoke and regression, but also to mark up different information like which team will own those test cases. [25]

8.2. Automated testing with BuildBot CI tool

To execute the test in an automated way there should be a tool to start the execution. For the experiment of automated test execution the Robot TA team decided to try a tool called BuildBot. Requirements for CI tool in the experimental phase showed that it is capable of getting the latest test data from SVN, executing RF commands, showing the latest execution report & logs and storing results for later use. Also, the way to trigger actions was considered and the tool needed to handle at least SVN polling and to be able to time schedule trigger option.

BuildBot is an open source CI tool licensed under GNU general public license and was started as a light-weight alternative for the Mozilla ThinderBox CI tool. [52, 53] It is built to automate compiling and testing phases to validate code changes of software builds. BuildBot free up developers from doing routine tasks to do more demanding and interesting ones, such as the actual development. Software build can be compiled and tested in parallel on a variety of platforms or projects can be made with any programming language, which makes BuildBot platform and language independent. BuildBot itself requires on Python and Twisted networking engine [54] and so host requirements are minimal. Results and reports of compilations or tests can be delivered via web-page, IRC, email or you can build your own protocol so

possibilities are limitless. This means notification of broken builds can be given to the responsible developer or stakeholder without human interaction. BuildBot gives the opportunity to track interactively progress of builds and provides remaining time estimation. It is easy to configure, because Python can be used in the configuration.

Robot TA team decided to take BuildBot as part of our automated FT test, because it was easy to configure, had no need for graphical user interface, needed to execute test in consecutive way, it is an open source product and it has quite a large user community [55]. The main requirements for CI tool like getting latest test data from SVN, storing and capability to show results reports and multiple ways to trigger tasks were found from BuildBot. In the regression testing experiment BuildBot was utilized by triggering task with time schedule, storing results to hosts disk and showing the results via BuildBot's www-server service. After a short experimental time, this arrangement turned out to be so successful, that the decision was made to take this as part of the production and official meter of development. Next phase was making automated smoke test tasks for BuildBot. In a similar way, those smoke test set task needed to upload test data from SVN and show results via www-service, long term result storing was not required, although no results clean up steps were designed or implemented either. These smoke test set tasks are only triggered manually by the user by pressing the "Force Build" button which will initiate a new Build, meaning new smoke test set task.

8.3. Manual functional testing

In rare situations, the fully automated test case creation is not feasible or even possible. One good example is physical damage or failure of some unit in SUT. This can be rarely emulated with test tools and is usually done by removing the unit or by power switch off. Another manual testing form is random exploratory testing [4, 22]. The exploratory testing is form done when results are not known beforehand, so testing cannot be easily made as test case scripts. In some way it can be thought as a defect or fault hunt, where the test engineer tries to find ways to cause some

unexpected behavior of SUT. Usually these tests require an experienced test engineer who has a lot of knowledge of SUT and its behavior. Due to the unexpected nature of the behavior in both of these testing forms, it is not feasible try to fully automate those kinds of test cases. This means that documentation of those test scenarios is even more important than for fully automated test cases, because all defects, faults and errors have to be reproducible before they can be accepted as faults. In exploratory testing this means that at least all steps from some zero points for example reboot have to be recorded somehow. In outage testing automation should be used as much as possible to ensure this reproducible requirement to be as easy as possible. In MME development, no manual test cases or scenarios are yet implemented, but in the near future, those are likely to become a reality which cannot be avoided.

9. End to end Continuous Integration

The final phase and goal for automated FT testing is complete CI pipe where all tasks after committing code to regression testing results is automated. This will enable the opportunity to execute all tests in a flexible manner, where changes will always trigger some tests and the latest software is under testing. Also, this enables more efficient usage of our test environments, more feedback and faster results. To achieve this goal, software projects must have all phases fully automated including unit or module tests, software configuration management (SCM) for creating software build, software installation and commissioning, smoke and regression test sets for functional testing. Even more extensive continuous integration can be achieved, if it includes automated stability and performance testing, system verification and network verification.

In MME development all phases preceding software installation were already automated before this thesis work was started. In previous phases from designing automation testing scheme to automated FT smoke and regression test sets, has given opportunity and readiness for complete end to end CI pipe. Only few things are still missing from complete CI pipe; automated software installation and commissioning and linking all phases together. Of course, if one would consider CI from a larger scope, it could also include fully automated stability and performance testing, system testing and network verification, but those are out of this thesis' scope.

For achieving automated software installation and commissioning the project already a used proprietary tool called Service Laboratory concept. With this tool new builds could be installed and commissioned to SUT automatically, without human interaction. The concept included some preliminary framework for CI testing, but not sophisticated enough for projects' CI needs. In spite of it, tool suited quite well to be taken as part of the continuous integration pipe, but some integration work and interface planning was still required.

After completing all of these tasks, the project would have complete end to end continuous integration pipe starting from the committing code to the results after completed regression testing. This would allow a round-the-clock testing flow, which would lead to a more effective usage of equipment and greater testing coverage. The main issues still were to decide architectural structure of the CI pipe and design and implement tasks for required interfaces between all tools needed in the process that did not exist yet. The architectural challenge was to decide, whether the CI pipe flow could be controlled by one of the highest level tools and be based on hierarchical model, or should it be implemented as flat just by chaining different tools together?

9.1. Service Laboratory concept

Service Laboratory concept (SerLab) is a NSN proprietary tool for automated software installation and commissioning to test equipment. SerLab takes advantage of 4Booking, an equipment reservation handling and register tool. Together these tools form an extensive equipment resource handling and software installation system, which can be used as part of the CI pipe. In a simplified manner the SerLab has three basic functions; gathering software build and equipment pool information, storage for hardware configuration and software installation macros and execution of those macros against software builds in certain environment. The 4booking tool has only two basic functions; management of item and pool information, and handling reservation information of equipment items.

One of SerLab's main functions is gathering software build and equipment pool information. When software builds are integrated and compiled at SCM, there will also be the generation of an xml-file, which contains information about the build. This XML-file contains information like build id, target platform and application and CI test related information. After compilation is done the build is copied from SCM to distribution servers with the XML-file which contains the builds' information. The SerLab gets information of compiled builds via these XML-files. The equipment pool information, containing identification information of pool and its items, SerLab

gets directly from the 4booking system. Pools are just logical containers for equipment items; like a MME or other network elements. With pools equipment items can be more easily classified in different groups and purposes. Based on pool's settings it can be used just for as a logical storage unit of manual testing equipment or SerLab can use pool's equipment in continuous integration for automatic software installation, commissioning and testing purposes.

The other significant main function of SerLab is acting as the place for hardware configuration and automated software installation macros. The significance doesn't come just from a role as storage place for those macros, but from the ability to also execute macros via SerLab. Each equipment pool has its own settings and all equipment items inside the pool have their own settings and configuration macros. In item's settings is stored basic information of the item, like IP-address, default route, subnet mask, and credentials etc. These can be used to test connectivity and in the case of new software build installation, making the basic configuration for that element. Besides just basic configuration, a modern network element needs an installation configuration which is referred as; commissioning an element and hence this kind of macro is called commissioning macro. After the basic settings have been given on the ATCA hardware, the only unit commissioned in the system is OMU and all other units have to be commissioned by executing the commissioning macro. This installation configuration macro contains information about the roles for the rest of the CPU blades and units attached on ATCA shelf. Macro also contains unit specific information like IP addresses, plug-in module information etc. After the execution of commissioning macro an element is ready to work, but it has default settings for everything. In modern telecommunication network each element should have a different kind of network, location, group, element etc. specific identification digits and other relevant information. This information is stored in hardware configuration macro which is the last macro that SerLab will execute, if all previous phases have been successfully executed. SerLab also stores logs and results of all macro executions and makes debugging and tracing possible, if needed.

The 4booking tool is a supportive tool for SerLab and is also used as a tool for handling equipment reservation information in manual testing. In continuous integration and a more specific SerLab service point of view, 4booking offer two main functionalities, the management of equipment information and handling reservation information of testing equipments. The management of equipment information can be divided into two parts; equipment pools and items. Pools can have multiple equipment items allocated in them, but equipment items can be allocated only in one pool at a time. For an equipment item one must define unique item id, name, purpose of use, business related information, responsible person, state, OS platform and application as equipment item's mandatory information. Id, name, purpose of use and responsible person fields are quite self-explanatory and all are free text fields, only requirement is that id has to be unique. Business related information is selected from dropdown menu and is meant for selecting the right business line, business units and their projects, so equipments can be searched and statistics made from the business angle. This promotes to mapping how many resources are used in certain project etc. State field contains information of item's status and can be Draft, Inactive, Active, Suspended, Flea market, Removed or Lost, being the most common states Active or Draft. The OS platform and application are fields indicating the appropriate software for the item. Out of these fields SerLab uses item id, OS platform and application information in selecting the right software build for each equipment item. Pool information contains the name, state, OS platform, application, SCM service, TWA service and pool content aka items allocated to the pool. Name field is self-explanatory and state field has the same purpose as the item's information, but values are different; Draft, Inactive, Active, Suspended and Removed. OS platform and application fields have the same values as in item's information, but are used in SerLab when continuous integration mode for pool is selected. In continuous integration mode SerLab will automatically install and commission new software build into one of the pools equipment items, if there is an item available. If there are no items available, new software build will go to queue and will be installed and commissioned when the equipment is free to be used. SCM service field is meant to indicate which commissioning service is used with the pool, and currently only SerLab is available. TWA service indicates the continuous testing

service to be used with the pool, the default and only value for this field is also SerLab at the moment.

The reservation management via 4booking for equipment items is implemented with simple graphical web interface for users or Simple Object Access Protocol (SOAP) [56] message interface for machine to machine communication. In web interface reservation can be made just by first searching equipment item by name and then clicking free slot from reservation calendar, which is seen in *figure 24.*

Figure 24: 4booking reservation calendar

This will trigger next pop up window where one must confirm and modify reservation length, if necessary.

After accepting reservation schedule by pressing ok, the last window will appear *figure 25.* In this window one can choose reason for Usage, possible choices are Personal, Calibration, Maintenance and Urgent maintenance, personal is the normal and default option.

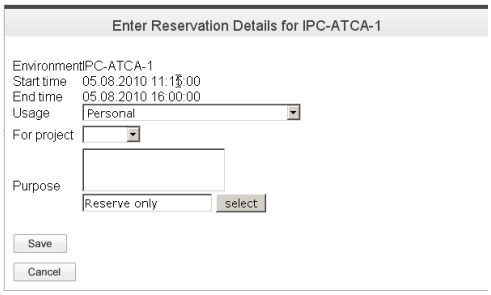

Figure 25: Reasoning and requested software build window

Also one must define which project to use and describe thef purpose as free text. The last field is reserved for selecting possible software build from dropdown menu, if needed, but default value is reserve only. If one would select some software build to be commissioned at the beginning of the reservation, 4booking would send request to SerLab which would start to carry out the execution of this request at the starting time of the reservation. In reserve only mode, equipment item is reserved for certain user and no commissions can be made during this period by other user via SerLab tool.

SerLab can be used to commission new software build into hardware in two modes. First one is fully automatic CI mode and the second one is the manual reservation mode. Mode of operation can be done by pool and from pool's settings in SerLab. *Table 3* describes both CI and manual reservation process flows in commissioning a new software build in simplified manner. SerLab will continuously poll for new software builds and add them to the queue; if there is a newer software build released before an older one is gone under commissioning, the newer one will take the first place in the queue and the older one will not be commissioned automatically anymore.

Step	Continuous Integration mode	Manual Reservation mode
1	SerLab notice a new CI build	Select free slot from 4booking reservation calendar
2	SerLab will check, if any pool with new build's platform and application is in CI mode	Confirm reservation time and length
3	SerLab will check, if any of the equipment items on pool are free	Give reasoning for reservation
4	SerLab will make reservation for a free item	Select a new CI build and commissioning macro
5	SerLab will execute commissioning and install the new software build into item with pre-selected CI commissioning macro	SerLab will execute commissioning and install the new software build into item with selected commissioning macro
6	SerLab will execute Smoke Test Set, if available	SerLab will inform user when commissioning is finished. Verdict can be Successful or Failed, if something goes wrong
7	Release CI reservation from 4Booking	Reservation in 4Booking continues until expired or canceled.

Table 3: The process flow for commissioning a new SW build

9.2. Complete continuous integration tube

Ideas behind complete continuous integration tube were quite simple; automate repetitious and ponderous tasks, increase visibility of testing, find defects and faults as early stage as possible and increase testing coverage and usage of testing equipment with automation. To achieve this all tasks from after committing code into

SVN to generating regression testing report has to be automated. From this thesis point of view tasks can be divided into two logical parts; before and after new complete software build. Before a new software build part contains module or unit compiling tasks, unit or module testing, request to add changes to new build and compilation of new software build by SCM. This part was already automated prior to this thesis work started. In thesis work's main challenge has been building test automation scheme for MME network element containing testing framework, design and implement automated functional testing and attach those to already existing pre-build systems. This has been achieved by running those tests in scheduled mode.

Beyond this thesis original scope, a future work for achieving complete continuous integration tube has been started. Those plans have two options for this tube a flat or a centralized architectural model. In flat architectural model all tools would communicate with next tool to trigger the next phase. Also from some phases results would be collected to some centralized service. In centralized architecture there would be a tool which would be considered as the highest level component. This component would trigger the next phase ongoing and collect the results after its execution. The tool would act as an organizer and result storage.

The flat architecture model is planned to have 8 different steps, 4 before build compilation and 4 post build compilation steps. All these steps and process flow can be seen in *figure 26*. All steps would trigger the next step, if step would be completely successfully executed. The content of each step is given just in a general manner and not in a detailed format.

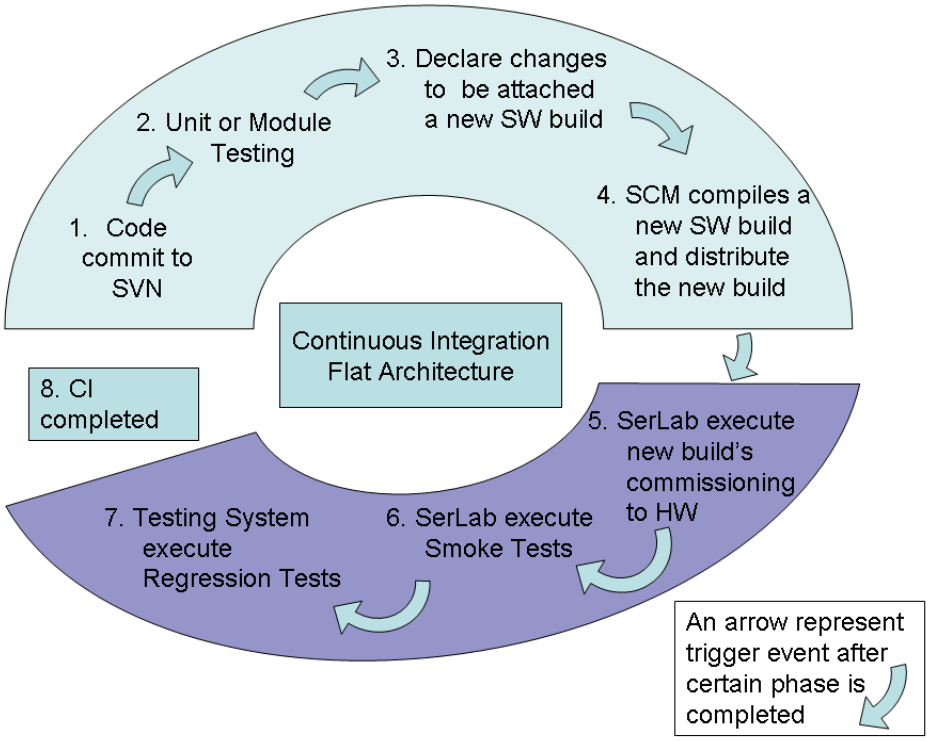

Figure 26: A Flat Architecture model in complete CI

The first step would consist of a manual part and it would trigger the next automatic functionality. The manual part would be checking the program code to SVN. Second step would be automatic unit or module testing, depending on the code or part of program and its test scheme. This step would store results of execution, successful or failure, or might send them to result service. Third step would be announcing successfully tested changes to complete build. Fourth step would be compiling a new software build, SCM tools would wait for change announcements for a while e.g. 10 minutes and if no further changes would occur, start to compile a new SW build. This waiting period is very feasible, because compilation of complete build can take quite a long time compared to module compilation. In some cases into a new build there has to be attached multiple modules at once to keep interfaces and cooperation of different modules intact. After successful compiling SCM tools would distribute the new build and its information to distribution servers, where proceeding tools can fetch it. Fourth step would be the last one, to complete the pre-build automation phases.

In the fifth step SerLab would notice the new build and initiate build commissioning into one of the CI pools equipment items. After commissioning would be ready, SerLab would internally trigger the sixth step and initiate Smoke testing. SerLab would store all results from commissioning and Smoke testing. Seventh step would be triggered by SerLab after successful Smoke testing. This last step would take the longest time and hence has to be executed separately from commissioning service to save resources. Simultaneously and parallel to the regression testing there could be also triggered network verification and system verification tests, because in smoke testing all the basic functionalities have been tested and verified to be correct. This would shorten overall testing time, if corrections or new build would be needed to be delivered to the customer as soon as possible.

The centralized architecture model, which can be seen in *figure 27*, is planned to have also 8 steps, but with slight modifications compared to the flat architecture. The first four steps would remain as the same as in the flat architecture, because there is no need to modify the already existing and well working system. The main difference would be the highest level tool, which would control and organize all steps proceeding step 4. Steps from 5 to 7 would be modified to use this highest level tool guidance in triggering and storing results. In the fifth step, SerLab would not try to notice or poll new builds from distribution servers, but would be triggered by the highest level tool, which would do the polling instead. After completing commissioning, SerLab would inform results of it to the control tool, instead of initiating smoke test execution. Then in the sixth step, the control tool would trigger smoke test to be executed by testing system and afterwards would get the results. The seventh step would be the same as the sixth, but the only difference would be execution of regression tests, instead of smoke test set.

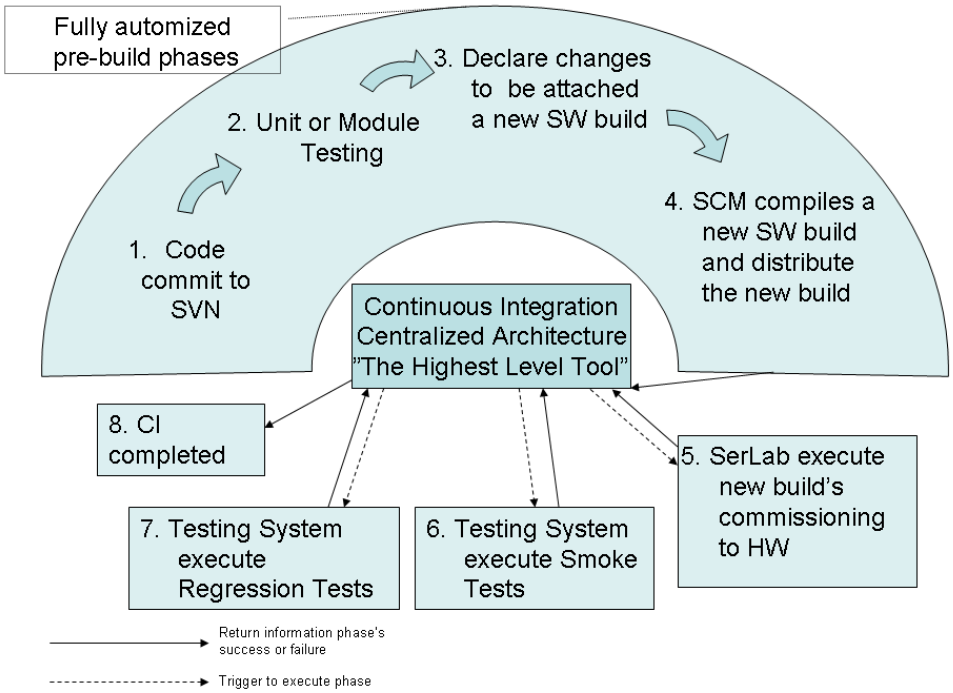

Figure 27: A centralized architecture for complete CI

Main differences between these two models are the number of needed interfaces and the different tools. Main advantage in flat architecture is the requirement for only one new interface between SerLab and the testing system. All other interfaces already exist. Disadvantage in the other hand is the dependency of tools, the pipe is as robust as its weakest link and it can be difficult to try to replace some non-working parts afterwards. For example, development and maintenance of SerLab is some other teams' responsibility, and changes and defect fixing can be difficult. Main advantage in centralized architecture is the weakness of flat architecture; all tools are quite easily replaceable, if needed. Disadvantages are the need for extra tools and extra interfaces. Here also one has to consider the work needed for maintenance work for changing interfaces, because tools always evolve and hence interfaces are quite certainly changed as well from time to time.

10. Results

Implementing the test automation scheme for LTE core network element can be considered to be successful, except for the end to end continuous integration part. The time window reserved for making this thesis was too tight and limited for achieving results on that final goal. Overall status nevertheless was a bit better than expected at the beginning due to the fact that other parts succeeded nicely and some parts even exceeded expectations.

The first practical task was to get information and learn the basics of Robot Framework test automation framework. After a short learning period of Robot Framework's architecture and its features, the development of the preliminary test library for the MME network element was started. This turned out quite well and Robot Framework was conclusively selected to be our test automation framework. Development for more sophisticated test libraries, such as test control libraries for LTE emulator and TTCN3 tester, was started right after Robot TA team members reached adequate skill level. All these preceding tasks succeeded and ease of development and usefulness of test libraries exceeded expectations. All test libraries have preserved without major need for refactoring and only a couple serious defects are found.

One of the main design principles for Robot Framework test library and test automation scheme was that all functional test cases should be fully automated. All design tasks took this as the first guideline and it had highest priority from all others. The principle was followed almost with all test cases, only two test cases were built without full automation, but automation level was over 80 % containing automated test case setup and teardown and most of test case functionality. There were two main cases in which test cases were not 100% automated. The first case was where hard coded values had to be used and it was infeasible to parameterize those values, because test case was needed only a couple times during one sprint. In other test case sudden removal of SCTP connection was needed and it was not feasible to make it as

software based inside the test tool with time window of one sprint. Nowadays test tools have been enhanced and there exists the possibility to do software based sudden connection removals.

To emphasize full automation test case goal a nightly regression testing started as an experiment which broaden as official meter of products stability and progress. Nowadays nightly regression testing is one of the key testing methods and visibility of build's sanity relies on it. Also smaller automated smoke test sets were built to help teams test their code changes easily with functional testing besides normal unit or module testing, giving quick feedback on how changes are working as integrated with other modules. Result of these automation features manual testing without Robot Framework is kept in very low level. Only some testing of test tool is carried out as a manual procedure.

Results of training and competence transfer to users also succeeded better than anticipated. Robot TA team members did not have any pedagogic training or background, but nevertheless, the learning curve of the new users has been remarkably high. The snowball effect, where one already trained user will share his information and knowledge with beginners, worked out very well.

The only part that was left unfinished was the implementation of end to end continuous integration pipe. Nevertheless planning and study tasks were successfully accomplished, but the final decision on how to proceed and the implementation of work accordingly was still missing.

Overall success rate was at least moderate, the implementation and design for the test automation scheme for LTE core network element can be considered as successful. During this thesis work and from beginning of the MME element development teams have gone through a big change, from waterfall model and its tools to agile model and the tools used with it. As for the testing point of view this change has to be graded successful.

11. Conclusions and Future Work

This thesis has described how test automation scheme for the LTE core network element was built and also presented the main requirements and pitfalls for such system. Nowadays, demand for faster, better and more comprehensive testing targets are the key driver for test automation and those targets are in many cases unachievable by only using manual testing methods.

The exact comparison between test automation and manual testing was not possible to accomplish, since all test cases were designed to be fully automatic, but the literature and studies imply up to a 40 % increase in velocity when using fully automated test cases versus manual test execution. The literature also suggests at first to invest more effort for test automation at the beginning of the development pipeline, such as in unit or module level testing. Only, when the first stages of development are sufficiently automated, the change of focus to next levels is feasible, because return of investment is higher at the beginning of the development pipeline.

In this thesis focus was put into the functional testing level, because all preceding testing levels were already fully automated in advance. In functional testing, the main challenge was to set a new mindset for all test engineers, in which all test cases should be automated and stored under a version control. The other considerable obstacle was to find a way to parameterize all the needed variables within test cases. All obstacles were solved and nowadays every new test case is done fully automated, if anyway feasible. Also, the atmosphere towards test automation has changed and its advantages have been noticed. The only setback was the extremely tight schedule for end to end continuous integration pipe, which would include also functional testing; unfortunately the implementation of this part of the thesis work was left as future work.

Future development after this thesis could be the completion of end to end CI pipe, which would increase the velocity of testing and give a quicker feedback to teams about the overall situation. Also, the automatic analysis of results could be further enhanced to handle the simplest defects and errors. This would really release the workforce potential, which is now used in wearisome test executions and simple analysis tasks. Finally, test automation development is a continuous task and is never complete until the whole project is finished; there is always something to improve.

References

[1] Lillian Goleniewski, Kitty Wilson Jarrett, *Telecommunications Essentials, Second Edition: The Complete Global Source,* Addison-Wesley Professional, *2006*

[2] Korhonen Juha. *Introduction to 3G mobile communications.* Arctech House, 2001

[3] Kreuer Dieter, *Applying Test Automation to Type Acceptance Testing of Telecom Networks: A Case study with Customer Participation,* 14th IEEE International Conference on Automated Software Engineering, 1999

[4] Patton Ron, *Software Testing, Second Edition,* Sams, 2005

[5] GSM Association, *Market Data Summary*, e-document, from: http://www.gsmworld.com/newsroom/market-data/market_data_summary.htm, [retrieved March 16, 2010]

[6] *HSPA to LTE-Advanced,* Rysavy Research / 3G Americas, September 2009, e-document, from: http://www.3gamericas.org/documents/ 3G_Americas_RysavyResearch_HSPA-LTE_Advanced_Sept2009.pdf, [retrieved March 16, 2010]

[7] Nabeel ur Rehman, et al. *3G Mobile Communication Networks*, e-document, from: http://www.asadasif.com/es/files/3g-report.pdf, [retrieved March 14, 2010]

[8] *General Packet Radio Service (GPRS): Service description,* 3GPP TS 23.060 V8.0.0 (2008-03)

[9] *Third Generation Partnership Project Agreement,* e-document, from: http://www.3gpp.org/ftp/Inbox/2008_web_files/3GPP_Scopeando310807.pdf, [retrieved February 22, 2010]

[10] *About mobile technology and IMT-2000*, e-document, from: http://www.itu.int/osg/spu/imt-2000/technology.html, [retrieved January 5, 2010]

[11] *About 3GPP2*, e-document, from: http://www.3gpp2.org/Public_html/ Misc/AboutHome.cfm/, [retrieved March 15, 2010]

[12] *UTRA-UTRAN Long Term Evolution (LTE) and 3GPP System Architecture Evolution (SAE),* e-document, from: anynomous ftp://ftp.3gpp.org/Inbox/2008_web_files/LTA_Paper.pdf, [retrieved March 15, 2010]

[13] *LTE,* e-document, from: http://www.3gpp.org/LTE, [retrieved March 16, 2010]

[14] Sesia S. et al., *LTE, The UMTS Long Term Evolution: From Theory to Practice*, John Wiley and Sons, 2009, ISBN 9780470697160

[15] *Requirements for Evolved UTRA (E-UTRA) and Evolved UTRAN (E-UTRAN),* 3GPP TR 25.913 V8.0.0 (2008-12)

[16] *General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access,* 3GPP TS 23.401 V8.7.0 (2009-09)

[17] *Nokia Siemens Networks Flexi Network Server: Ushering network control into the LTE era,* NSN customer documentation, from: https://www.online.nokia.com/, [retrieved March 25, 2010]

[18] *Evolved General Packet Radio Service (GPRS) Tunnelling Protocol for Control plane (GTPv2-C),* 3GPP TS 29.274 V8.0.0 (2008-12)

[19] *AdvancedTCA Q & A,* e-document, from: http://www.picmg.org/pdf/AdvancedTCAQA.pdf, [retrieved April 10, 2010]

[20] *Beizer, B., "Software Testing Techniques", 2nd edition*, International Thomson Publishing, 1990.

[21] Haikala I., Märijärvi J., *Ohjelmistotuotanto,* Talentum Media Oy, 2004

[22] Lisa Crispin, Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams,* Addison-Wesley Professional, 2008, ISBN 978-0-321-53446-0

[23] *IEEE Standard for SoftwareVerification and Validation*, IEEE Std 1012-2004, 2004

[24] McConnell, Steve. *Code Complete* (2nd ed.). Microsoft Press, 2004 . ISBN 0-7356-1967-0.

[25] *Robot Framework User Guide Version 2.1.2,* e-document, from: http://robotframework.googlecode.com/svn/tags/robotframework-2.1.2/doc/userguide/RobotFrameworkUserGuide.html, [retrieved December 14, 2009]

[26] Holmes A., Kellogg M.. *Automating functional tests using Selenium.* Agile Conference, 2006

[27] Tijs van der Storm. *Continuous Release and Upgrade of Component-Based Software*. Proceedings of the 12th international workshop on Software configuration management. 2005

[28] Cohn Mike. The *Forgotten Layer of the Test Automation Pyramid.* e-document, from: http://blog.mountaingoatsoftware.com/the-forgotten-layer-of-the-test-automation-pyramid, [retrieved March 25, 2010]

[29] Korkala M., Abrahamsson P., *Communication in Distributed Agile Development: A Case Study*. 33rd EUROMICRO Conference on Software Engineering and Advanced Applications. 2007

[30] Using *scrum in a globally distributed project: a case study*, Software Process: Improvement and Practice, Volume 13 Issue 6, John Wiley & Sons, Ltd., 2009

[31] Boehm B., *A Spiral Model of Software Development and Enhancement.* ACM SIGSOFT Software Engineering Notes. 1986

[32] *Manifesto for Agile Software Development*, e-document, from: http://agilemanifesto.org/, [retrieved September 13, 2009]

[33] Krasteva I., Ilieva S., *Adopting an agile methodology: why it did not work*, Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral. 2008

[34] Ferreira C., Cohen J., *Agile Systems Development and Stakeholder Satisfaction:A South African Empirical Study.* Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries. 2008

[35] Sengupta B. et al., *A Research Agenda for Distributed Software Development*. Proceedings of the 28th international conference on Software engineering, Shanghai, China. 2006

[36] Fowler M., *Continuous Integration*, e-document, from: http://www.martinfowler.com/articles/continuousIntegration.html, [retrieved November 13, 2009]

[37] Holck J., Jørgensen N., *Continuous Integration and Quality Assurance: a case study of two open source projects*, Australasian Journal of Information Systems, 2004

[38] Duvall P. M. Et al., *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley Professional, 2007, ISBN 978-0-321-33638-5

[39] Hill J. H. et al., *CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments*, 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2008.

[40] Sauvé J. P. et al. EasyAccept: A Tool to Easily Create, Run and Drive Development with Automated Acceptance Tests. Proceedings of the 2006 international workshop on Automation of software test. 2006

[41] Korpela J., Tab Separated Values (TSV): a format for tabular data exchange, e-document, from: http://www.cs.tut.fi/~jkorpela/TSV.html, [retrieved March 16, 2010]

[42] *Python Programming Language*, e-document, from: www.python.org, [retrieved December 14, 2010]

[43] *Robot Framework*, e-document, from: http://code.google.com/p/ robotframework/,[retrieved March 25, 2010]

[44] *What is Pydev?*, e-document, from: http://pydev.org/, [retrieved February 5, 2010]

[45] *About the Eclipse Foundation*, The Eclipse Foundation, e-document, from: http://www.eclipse.org/org/, [retrieved March 22, 2010]

[46] *Apache Subversion*, e-document, from: http://subversion.apache.org/, [retrieved September 23, 2009]

[47] Cordeiro L. et al. *An Agile Development Methodology Applied to Embedded Control Software under Stringent Hardware Constraints.* ACM SIGSOFT Software Engineering Notes. 2008

[48] TTCN3.org, *TTCN3 language* , e-document, from: http://www.ttcn-3.org/home.htm, [retrieved March 14, 2010]

[49] Wikipedia, *Telelogic*, e-document, from: http://en.wikipedia.org/wiki/Telelogic, [retrieved January 05, 2010]

[50] *Telnet Protocol Specification*, RFC 854, e-document, from: http://tools.ietf.org/html/rfc854, [retrieved February 05, 2010]

[51] *Port Numbers*, IANA, e-document, from: http://www.iana.org/assignments/port-numbers, [retrieved March 16, 2010]

[52] *BuildBot Manual 0.7.12*, e-document, from: http://djmitche.github.com/ buildbot/docs/0.7.12/, [retrieved March 15, 2010]

[53] *Tinderbox*, Mozilla Foundation, e-document, from: https://developer.mozilla.org/ en/Tinderbox, [retrieved March 14, 2010]

[54] *Twisted*, e-document, from: http://twistedmatrix.com/trac/, [retrieved March 17, 2010]

[55] *BuildBot SuccessStories*, e-document, from: http://buildbot.net/trac/ wiki/SuccessStories, [retrieved March 17, 2010]

[56] *SOAP Specifications*, World Wide Web Consortium, e-document, from: http://www.w3.org/TR/soap/,[retrieved March 16, 2010]