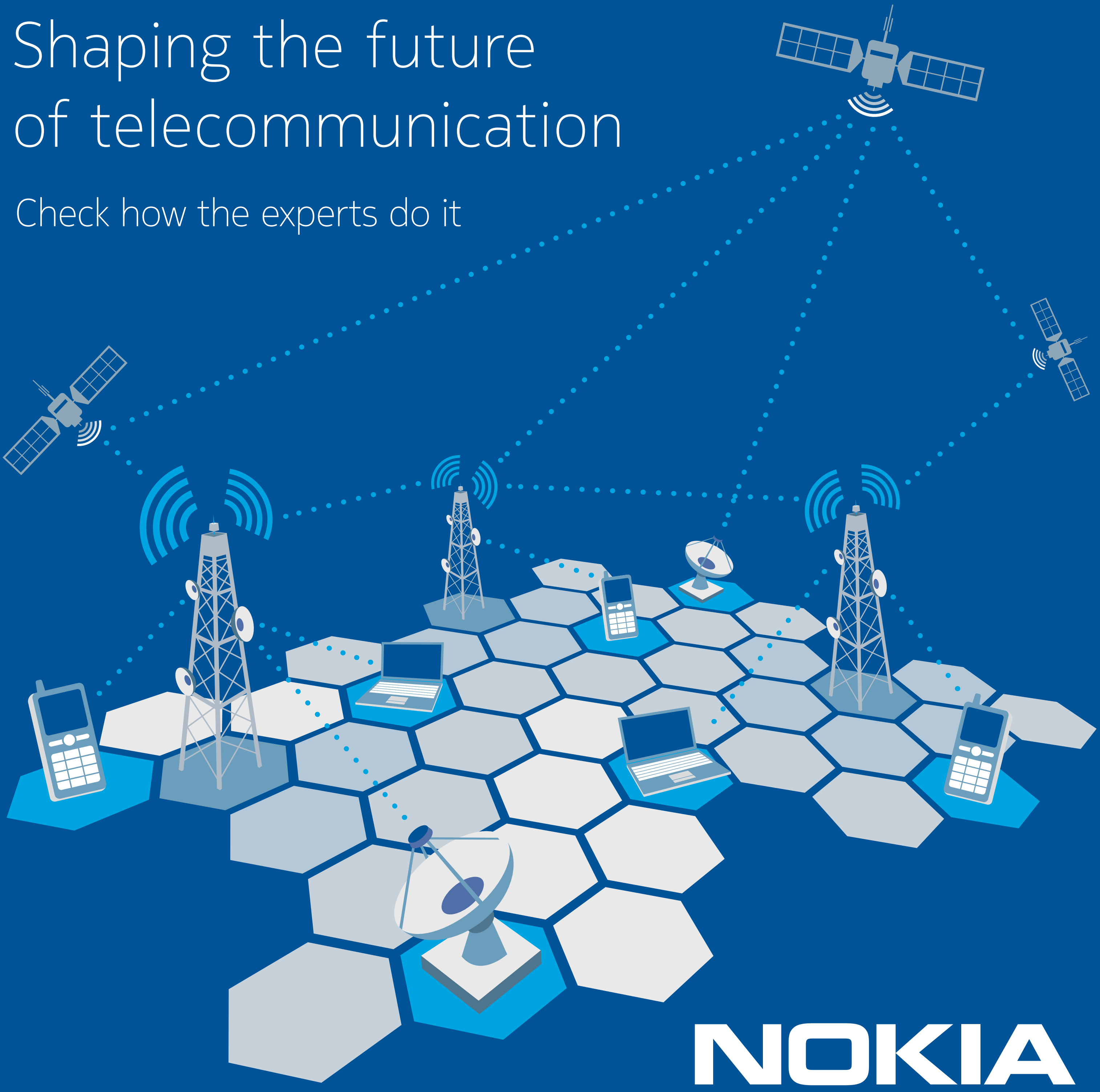
Shaping the future of telecommunication

Check how the experts do it



NOKIA

Dear Readers,

I am thrilled to present to you another joint effort of employees of Nokia Networks European Software and Engineering Center in Wrocław: “Shaping the Future of Telecommunication. Check How the Experts Do It.”

This collection of 23 interesting writings extensively covers four aspects of the programmer's work, including Advanced Technologies, System Engineering, System Development, and Best Coding Practices. The book takes a broad view of current trends in programming, and is of great educational value due to its accessible language and real-life case studies. Therefore, I am very pleased to introduce you to this anthology, and thank the authors for their passion and willingness to share, hard work, and outstanding contributions.

I wish you a pleasant read,

Bartosz Ciepluch
Head of Nokia Networks European Software
and Engineering Center in Wrocław

Advanced Telecommunication Technologies

Advanced Telecommunication Technologies

On Telco Cloud Edge: FlexiCMD Contribution into Nokia’s Concept for NFV

Przemysław Szufarski
System Product Manager
MBB Liquid Core

Introduction

In recent 30 years, there have been but a few technology turnarounds which are also cornerstones of mobile communications. The capability to migrate from analog to digital communication resulted in a GSM revolution in the 90s, migration from circuit-switched to packet-switched networks, and all-IP networks were a prerequisite for a Smartphone revolution in 2007. I believe that Telco Cloud technologies may have a similar impact on the future: opening network capabilities for a new user experience.

1. Telco Cloud and its environment

Telco Cloud standardization is driven by the Network Function Virtualization Industry Specification Group (NFV ISG) within the European Telecommunications Standards Institute (ETSI). The standard defines Virtualized Network Functions (VNF) and related supporting components (see **Figure 1**).

The standard introduces a new class of components (NFV Orchestrator, VNF Manager, and Virtualized Infrastructure Manager) dedicated to cloud management. Those components are expected to support cloud’s flexibility.

The standardization is setting a blueprint for network layout, but it is only one of many triggers for network migration. Moreover, the network may never reach the blueprint layout which was a target for migration! Interworking of existing network functions and new VNFs is a challenge which needs to be taken into careful consideration from the very beginning of cloud introduction. Obviously, an adaptation to the legacy network is up to Cloud-based VNFs, new components in the Communications Service Provider (CSP) network.

2. VNF over NF

The key components of the NFV architecture are Virtualized Network Functions (VNFs), replacing existing “bare metal” network nodes. Those components replace hardware-based components such as Call Session Control Function (CSCF), Mobility Management Entity (MME), or Charging Gateway Function (CGF).

However, NFV gains over the classic approach are mainly due to features which can be realized only with both components: the NFV architecture (see **Figure 1**) and related features of a specific VNF. The e2e architecture covers interworking of all network domains,

Figure 1 ETSI NFV scheme.

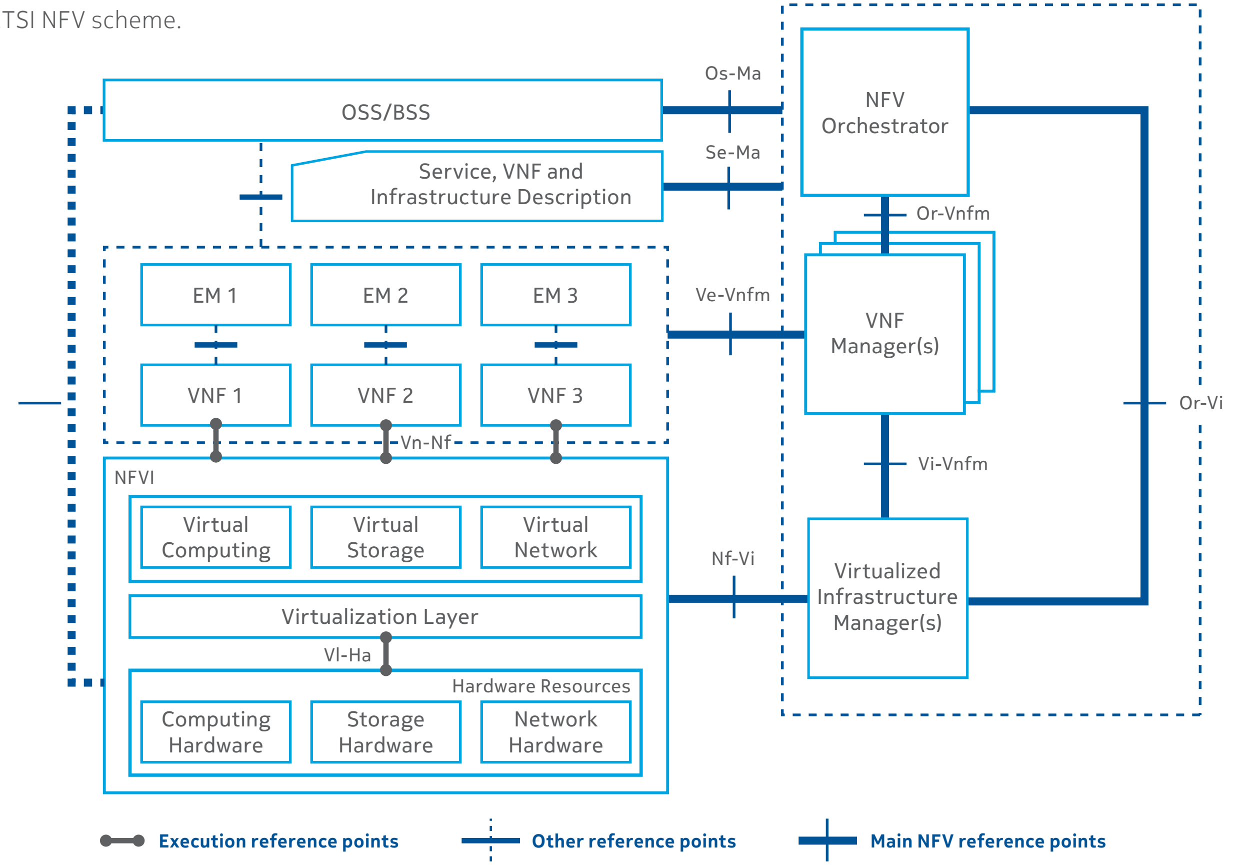
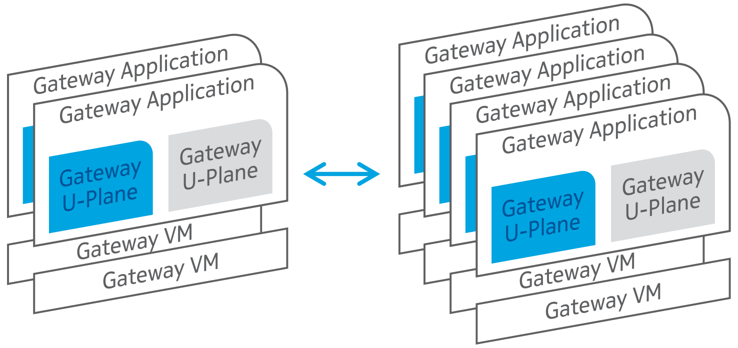


Figure 2 Example of VNF elasticity.



ensuring integration of legacy networks with VNFs. Virtualized Network Functions with Cloud Management and an Orchestration layer provide new functions to the CSPs. The fully cloudified VNFs would ensure a CSP competitive advantage due to the following features:

- Elasticity: where VNFs' capacity may be adapted to increased or reduced capacity demands
- Orchestration: where VNFs' cloud resources and interfaces are adapted by the Orchestrator according to current services
- Hardware-Software Lifecycle separation: more frequent software upgrades can be separated and become independent of a hardware upgrade, and hardware-related activities do not harm existing services provided by VNFs
- Automated Software Lifecycle: where VNFs are upgraded to new software version and enriched with new functions
- Hardware Independence: VNF could be launched on any hardware being part of the cloud

Merely the few features mentioned above are opening new opportunities for CSPs. For example, instead of an independent lifecycle of hundreds of hardware (HW) components, there is just one HW line to be looked after. Among many other benefits, HW standardization is the most obvious one. A better utilization of HW resources due to elasticity and orchestration allows increasing savings of the cloud-based infrastructure. With the NFV concept, the capacity overhead required to ensure service availability may be significantly reduced as well and quickly adjusted to current needs due to orchestration and elasticity functions. In general, the infrastructure will be better tailored and more adaptive to the end-user's and CSP's needs. With such tools, CSPs will be able to reconfigure their networks to optimize capacity for an actual mix of use cases (e.g. increasing VoLTE traffic) in the blink of an eye.

NFV features with a very high potential are Automated Software Lifecycle together with Service Orchestration. Those may reduce the innovation cycle from months to days or even more and automate it. Such a foundation allows CSPs to build innovative services in a similar way to Smartphone vendors by providing a platform for independent developers, with a capability to program network service, deploy and verify it with end users. Commercialization models are not yet available, but we may learn very quickly from the existing models of Smartphone Apps how to build communities around mobile carriers.

3. Facing the consequences of cloud orchestration

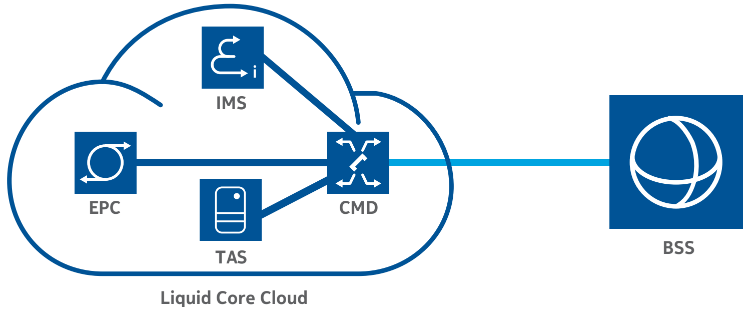
Orchestration allows streaming cloud capacity to dedicated services. It enables growth and termination of services in a much shorter period of time and with lower average resource utilization. This is achieved by a dynamic allocation/release of cloud resources to VNFs and a capability for VNFs' re-configuration. With this function, the cloud will adapt the network to the needs of services, optimizing and re-configuring it within days instead of months. These are the expectations shared both by CSPs and Infrastructure Providers at the Mobile Word Congress in Barcelona in 2015. Examples of scenarios for cloud orchestration:

- traffic transition between VNF instances, graceful termination of a VNF instance
- traffic split/offloading from one VNF among other VNFs
- network reconfiguration, deployment of new VNF instances within the network

What are the consequences of the orchestration in a cloud for legacy networks? As presented in the example in Figure 2, the number of Network Gateways may be increased or decreased based on orchestration requests.

Both scenarios, the expansion and reduction of VNF instances, are not expected to be performed on a daily basis in "bare metal" systems and, therefore, require special handling. In such a case, a smooth conversion of VNF interfaces needs to be foreseen via a Software Defined Network (SDN), load balancing, or mediation supporting VNFs' interfaces.

Figure 3 High-level concept for signaling reduction in IF charging.



4. Paving the ground towards NFV

The NFV concept is a quite fresh idea for Telco operators. It needs to be pointed out that the NFV standardization is in a very early stage in mid-2015, and there are many areas not covered by the standard yet. The very basic assumption is that standards provide a capability to use the equipment of different vendors together. It is believed that the measure of VNF concept's maturity could be the number of public references where different Telco Infrastructure Providers deployed their VNFs on a common NFVI. Until now, the number of such references has been very limited, if existing at all.

Nokia has started first commercial deliveries of Virtualized Network Functions and launched a number of demonstrations and trials to present our leadership in implementation of the NFV concept in general. Nokia took a bold step and is turning an entire portfolio of our Mobile Core products into VNFs.

Flexibility of Virtualized Network Functions requires special attention on the cloud edge, where flexible interfaces are terminated. Service Orchestration, Software Defined Network, and Virtualized Network Functions are the terms which in reality may be applicable to a part of the CSP network only. A proper solution to the transfer of Telco Cloud traffic to legacy components shall be introduced as part of network's modernization. Nokia's solution to a seamless integration of cloud and non-cloud environments, especially the Core Network (CN) and Business Support System (BSS) domains, is Flexi Converged Mediation Device (CMD). The design of FCMD has been adapted to ensure interworking with cloud and non-cloud network components.

5. Unified charging interface of Liquid Core

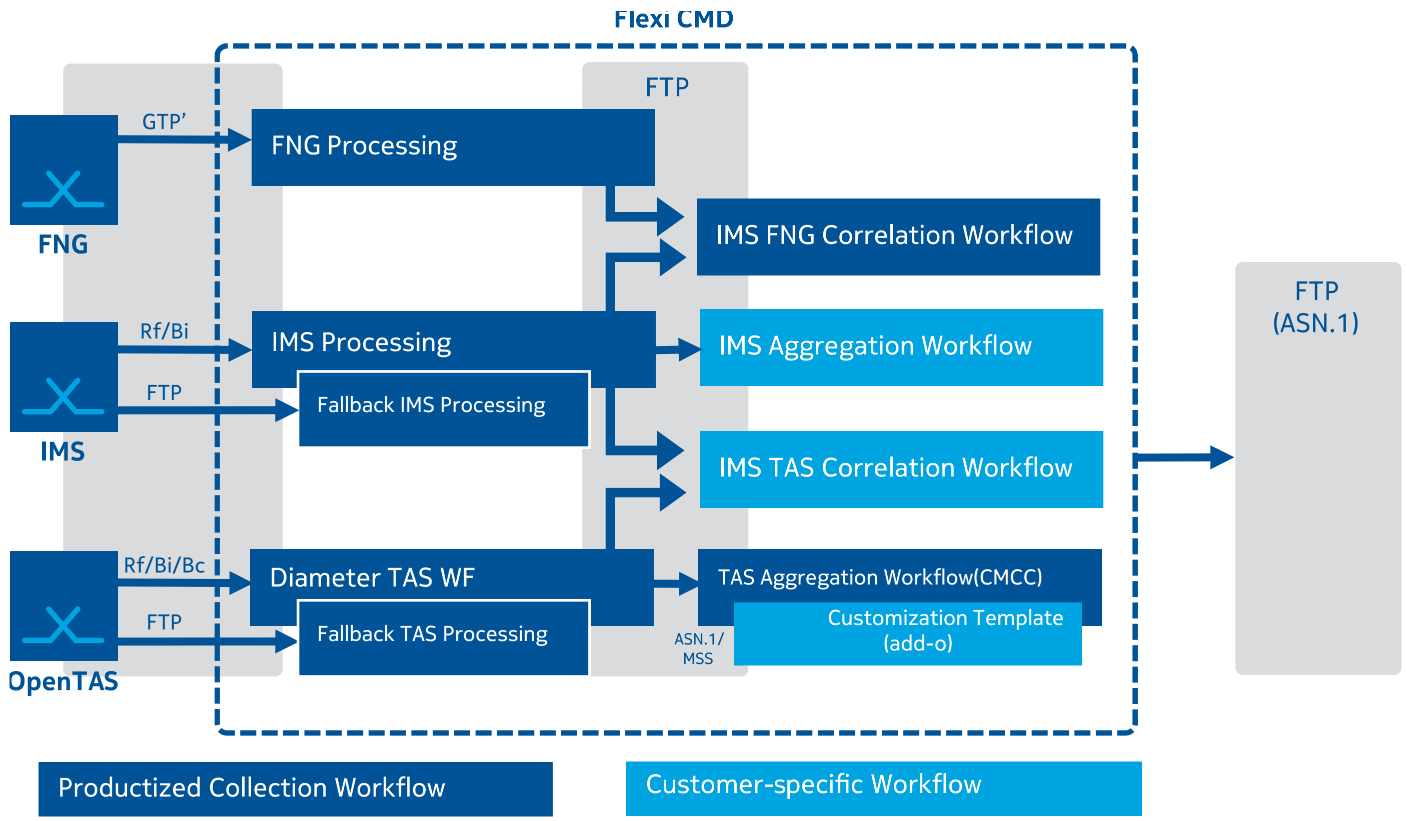
The Flexi Converged Mediation Device (FCMD) is used for integrating mainly the CN and BSS domains. It provides converged mediation capabilities such as the Unified Charging Gateway Function or Charging Data Function with possible extensions: Charging Gateway Network Address Translation (CGNAT) logs collection, exchange of roaming CDRs, integration with revenue assurance/fraud detection.

The main advantage of FCMD is deployment as VNF, part of the Telco Cloud. This allows to ensure proper handling of Telco Cloud specific technologies, i.e. elasticity and orchestration; FCMD is capable to reduce charging the data stream towards BSS (aka BSS offloading) via aggregation and correlation functionalities. Additionally, it is capable to act as a buffer between Core and BSS domains. These functions enable FCMD to cope with the mismatch between cloud and non-cloud network parts. Remaining VNFs may use FCMD as a default, pre-integrated destination point of charging interfaces, which results in a simplification of the VNF's architecture and a more flexible Telco Cloud–BSS integration. Another opportunity for an FCMD deployment in the cloud is a traffic correlation capability between VNFs, e.g. S/PGW and TAS, which results in a reduction of the amount of data sent towards BSS (reducing Billing System's capacity requirements) and provides additional extended information about subscriber's usage collected and correlated from different network elements. A high-level concept of the solution is presented in Figure 3.

An FCMD internal concept for correlation of charging data streams is based on a definition of FCMD internal workflows. Workflows dedicated to a particular VNF are able to receive, aggregate, and standardize the format of data charging. Then, an additional layer of workflows allows reducing the amount of outgoing information by correlating the data streams from selected VNFs. A workflows map for offline charging aggregation and correlation between Flexi NG, IMS, and Open TAS is presented in Figure 4.

To ensure proper handling of performance, availability, elasticity, and orchestration requirements, the FCMD deployment will be based on multiple instances of described workflows. When an internal FCMD interface between workflows is ready for that, the immediate challenge becomes the integration with external data sources. FCMD has been pre-integrated on a product level with leading Liquid Core VNFs: Flexi NG Cloud, IMS Cloud, and Open TAS Cloud. An ex-

Figure 4 FCMD workflows map for traffic correlation and aggregation.



ample of the interface between FCMD Cloud and Flexi NG Cloud is presented in **Figure 5**.

In the example, the Flexi NG registers each FCMD's workflow as an independent Charging Gateway. Each FNG Node distributes charging data based on a sticky round robin algorithm (traffic is evenly distributed, but a single session is sent to the same instance) to all available FCMD workflows. In case of an extension with FNG Nodes, a list of valid FNG Nodes is updated on FCMD. In case of new FCMD Processing VMs, a list of valid FCMD Processing VMs is updated on the FNG. This ensures automatic handling of orchestration, elasticity, and VM recovery (availability) scenarios.

6. Conclusion

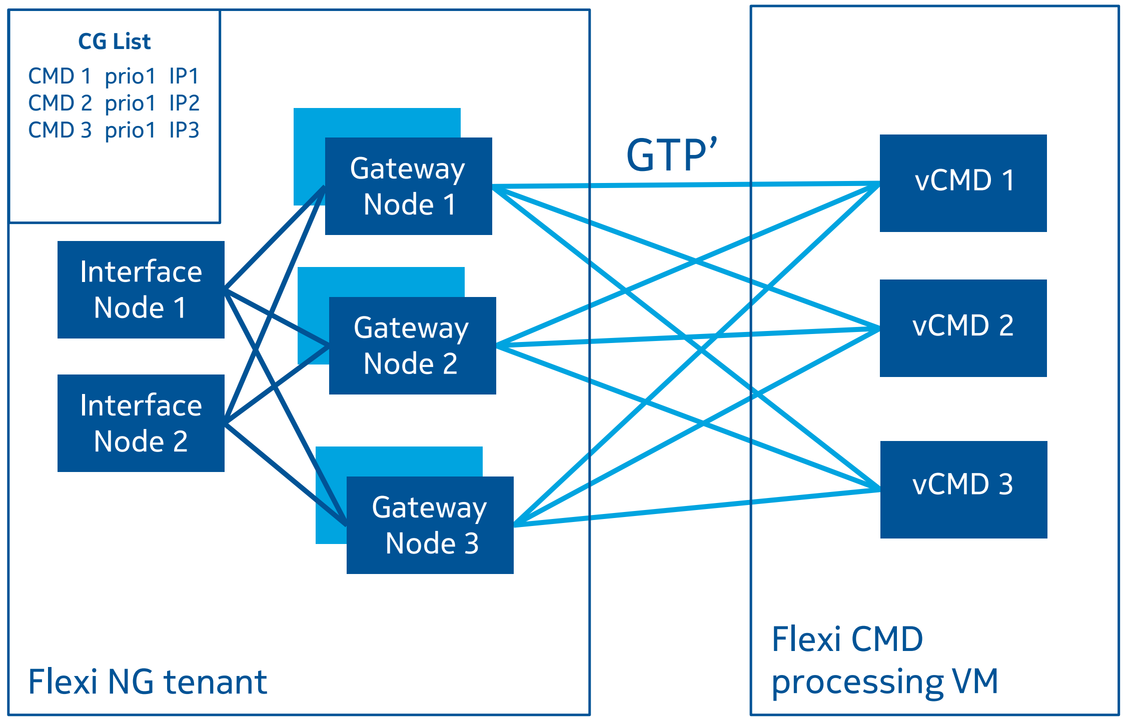
Nokia's FCMD component is an excellent solution to integrate Telco Cloud with the existing Business Support System. It has been pre-integrated with Nokia's VNFs and is ready for further integration based on Customer needs.

FCMD Cloud 16 is an initial FCMD release integrated with Liquid Core Cloud, Nokia's concept for Telco Cloud. FCMD 16 has all benefits of Telco Cloud integration, i.e. automated deployment and high availability. It has been pre-integrated with Nokia's VNFs: Flexi NG 16, IMS 16, and Open TAS 16 and supported by Nokia's Telco Cloud components: NCI O16 and CAM O16.

References

[1] Nokia,TelcoCloudManagement:http://info.networks.nokia.com/telco_cloud_management_lp.html
[2] ETSI GS NFV 002, Network Functions Virtualisation (NFV); Architectural Framework
[3] ETSI GS NFV 003, Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV
[4] Nokia, Reinvent FCMD for the Cloud, D496975900

Figure 5 Connectivity between VNFs: FCMD – FlexiNG.



About the author

I am a graduate of Electronics, Telecommunications & Informatics faculty of Gdańsk University of Technology. I work as a System Product Manager in MBB Liquid Core Technical Management. In Liquid Core, we build future Core Networks, which are delivered faster and are more flexible. The main responsibility of the Technical Management team is to transform innovative ideas into enterprise class software products; it is a both challenging and inspiring role.

Przemysław Szufarski
System Product Manager
MBB Liquid Core

Cloud Security – Risks Brought by Virtualization

Bartłomiej Dabiński
Security Solution Engineer
MBB Security

Marcin Otwinowski
Integration and Verification Engineer for Security
MBB Security

NOKIA

1. Introduction to cloud virtualization

Virtualization is a technology that abstracts logical resources (computing, storage, and networking) from physical hardware. It enables highly flexible, efficient, and scalable system architectures that offer instant creation of differentiated and personalized services. There is no doubt that virtualization has turned out to be the market disruptive technology and has also been the enabler of cloud computing, another revolutionary business concept.

NFV (Network Functions Virtualization) is an approach proposed by ETSI (European Telecommunications Standards Institute) that aims to deliver a high-level specification of the standardized architecture of virtualized telco networks. The fundamental goal of NFV is to consolidate many network equipment types onto industry standard servers, network nodes, and storage, which could be located in a variety of network PoPs (Points of Presence) including datacenters and end user premises [1]. This enables the abstraction of a standard, virtualized environment for operating VNFs (Virtual Network Functions).

While traditional networks rely on fragmented vendor-specific hardware, in the NFV approach network functions are implemented in software that can run on generic industry standard servers. Deriving hardware from software enables independent upgrades of each layer of the system, simplifies scaling, and brings elasticity in various load conditions because the software can dynamically be moved to another location.

2. NFV infrastructure overview

The NFV architectural framework shown in Figure 2 specifies the functional blocks and the interfaces between them. The roles of the blocks and their interconnections are briefly described below. The components that are the main focus of NFV are MANO (Management and Orchestration) blocks and NFVI (Network Functions Virtualization Infrastructure). The other components (OSS/BSS, EMs, NFs) are available in present deployments and as such they are external to the NFV perspective. Nevertheless, the interfaces between them and the NFV components are in the scope of ETSI works.

Figure 1 Comparison of a traditional and virtualized networking approach [2].

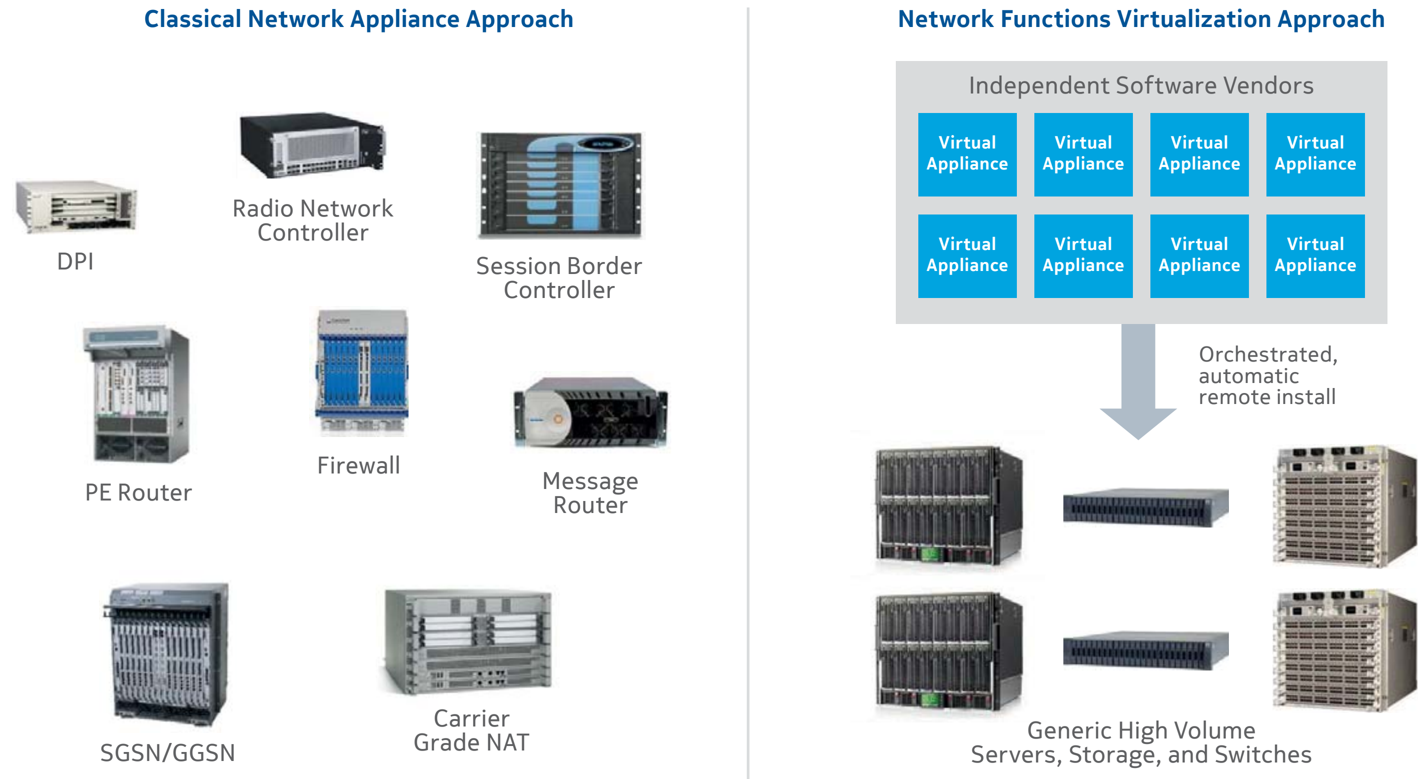
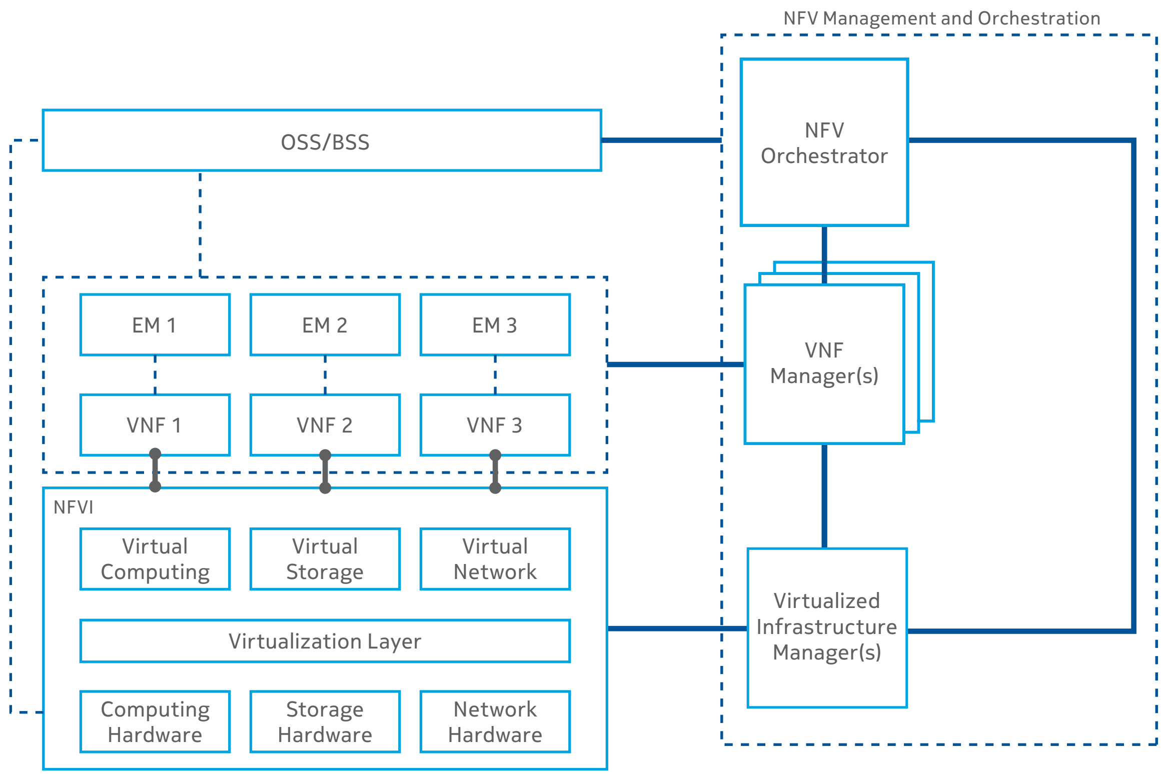


Figure 2 NFV reference architectural framework [6].



OSS/BSS (Operations Support Systems/Business Support Systems) – computer systems used by telco providers for functions such as performance management, fault management, network configuration (OSS components) and payment management, order management, revenue management, and customer management (BSS components) [3][4].

EM (Element Management) – performs typical management tasks and supports the abstraction of the network elements implementing VNF(s). EM may also maintain statistics, logs, and other data concerning network elements [5].

VNF (Virtual Network Function) – is a concept representing the equivalent of a network node or a physical appliance in a traditional network. VNF usually consist of one or many virtual machines running on NFVI [7].

NFVI (NFV Infrastructure) – all hardware and software components that build up the environment in which VNFs are deployed, managed, and executed. From the VNF perspective, NFVI forms a single entity providing the VNF with resources (computing, storage, and networking) [6].

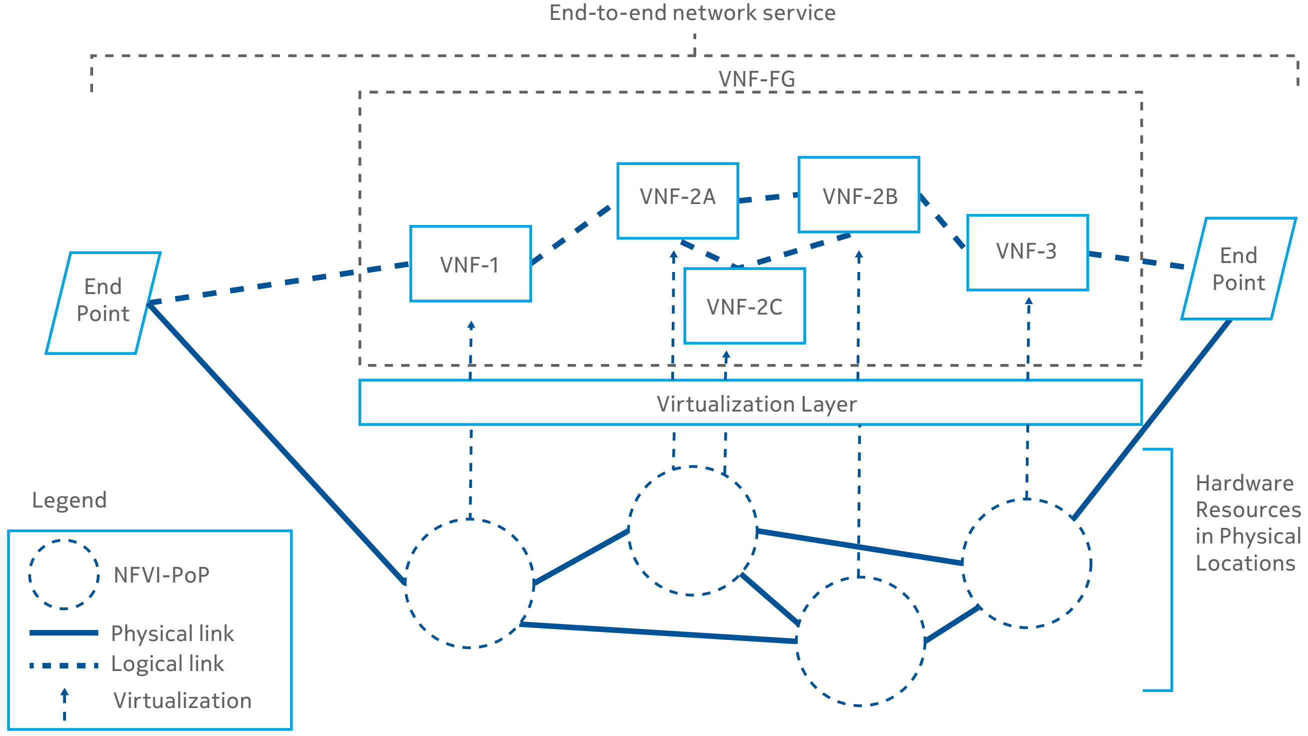
NFV MANO components:
NFV Orchestrator – manages the network services' lifecycle, supports and coordinates the operation of VNF Manager and VIM to ensure an optimized allocation of the resources [7]. CND (Cloud Network Director) is a Nokia product implementing the NFV orchestration functions.

VNF Manager(s) – is responsible for VNF lifecycle management (e.g. installation, update, query, scaling, and termination). For different groups of VNFs, multiple VNF Managers may be deployed [6]. CAM (Cloud Application Manager) is a Nokia commercial VNF Manager.

VIM(s) (Virtualized Infrastructure Manager(s)) – controls and manages the interaction of VNFs and NFVI resources. VIM collects information for capacity planning and allocates VMs with required computing, storage, and networking resources. VIM monitors the performance and health status of VMs [8].

VNF Forwarding Graph (VNF-FG)
A forwarding graph (also referred to as a service chain) is an abstraction of a network service. Nodes of the graph are individual VNFs, VNF sets, or other (nested) VNF-FGs. Graph edges represent data flows between VNFs. The links can be unidirectional, bidirectional, multicast, and/or broadcast. Figure 3 shows a simple example of an end-to-end service. The constituent VNFs can include a load balancer, firewall, CDN network etc. The end point devices, which are usually customer-owned equipment e.g. mobile phones, are outside the scope of NFV.

Figure 3 A chain of VNFs is a simple example of VNF-FG [6].



3. NFV from a security perspective

The ETSI NFV model does not include any security elements by default. The next chapters indicate that it should be extended in the future by security components. This paper focuses on security risks and aspects brought on by the virtualization of network functions. Threats generated by specific virtualization environments (e.g. recently discovered buffer overflow related VENOM vulnerability in QEMU FDC controller) are outside the scope of the paper. General security aspects of network elements (DoS protection, routing security etc.) are also beyond the scope. The paper indicates new security vulnerabilities that should be taken into consideration when combining generic virtualization vulnerabilities with generic network vulnerabilities. Nokia is currently developing its own security solution for virtualized environments. It is called Cloud Security Director and is complementary to the ETSI NFV model.

First of all, there is a need to build a profile of a potential attacker. Nowadays, generally, we can categorize them into three main groups [2]:

- group 1: criminal members of the general public (including former and current employees of network operators),
- group 2: coordinated small groups, such as criminal gangs or terrorist organizations,
- group 3: coordinated large organizations with a political, religious, or commercial agenda, e.g. government agencies, corporations.
- From the perspective of NFVI, different dimensions of attacks can be observed. The following are some examples as related to the aforementioned three categories [2]:
- end customers of retail network operators (applicable to all three groups),
- retail network operators (in most cases group 2 fits here in terms of industrial espionage),
- wholesale network operators (groups 2 and 3),
- hypervisor operator – usually the same party as either the infrastructure operator or the wholesale network operator (groups 1 and 3),

- infrastructure operators – the party that operates the computing, storage, and infrastructure network (groups 1 and 3),
- facility manager – the party that secures the physical building, racking, power, cabling, etc. (group 1).

4. Vulnerabilities analysis and techniques to fix them

First and foremost, vulnerabilities must be properly identified and analyzed. Otherwise, one can consider that virtualization in fact is just another layer of software that emulates hardware, and both of them (emulated hardware and software that performs emulation) need only to be patched to secure the system. In this chapter, however, it is assumed that virtualization is actually a separate layer that brings new challenges.

Complexity of environment

One such challenge is strictly connected to complexity. It is easy to imagine a very complex environment where physical and virtual layers of a network interfere and influence each other, especially when not a private virtual cloud but a shared one is considered. Poor planning and infrastructure deployment (both physical and virtual) can imperceptibly lead to a loss of control over the environment. This can then lead to accidental access to company-sensitive data across the shared cloud infrastructure. There is a significant threat here because some companies may actually try to perform industrial espionage activities. Therefore, an environment cannot be assumed secure unless the exact configuration of physical and virtualized layers is known and the resources operator is trusted. Moreover, unknown configuration cannot be properly verified. It is a matter of uttermost importance since an incorrect configuration, in an extreme case, can lead to a situation in which accidently (or maliciously) the entire datacenter is deleted by a single command.

Things should be as simple as possible, but not simpler – Albert Einstein used to say. Yet he probably did not have telco cloud environments in mind, this rule is very purposeful in this area. Correct planning and designing of such an environment is crucial for the future operation and secure separation of each virtual space. A relevant example here is the separation of the management network. Physical separation is favored in security aspects, but it could be prohibitively expensive to implement such a solution in real life. Therefore it is usually a trade-off between maximum security and a reasonable price (a combination of physical and logical separation). A general recommendation is that everything which is stored in a cloud should be carefully selected and well protected. Rotation and shredding mechanisms for old data should also be planned.

Secured boot and runtime

The next security issue is related to trust. Especially when access to a physical infrastructure is not provided or hypervisors are under a third party's control. A network operator has to trust a hosting provider sufficiently to run VNFs on the provider's virtualization platform. Respectively, the provider would rather be sure that the VNFs are genuine.

Nonetheless, as it is commonly believed that control is more efficient than trust, there are mechanisms that verify the authenticity and integrity during a booting process [2]:

- local attestation
- remote attestation
- attribution
- authenticity
- configuration management
- certificates
- cryptographic keys
- digital signatures
- hardware-specific features

Some good examples of technologies that can be used here are TPM (Trusted Platform Module), which gives the benefit of hardware built-in cryptographic algorithms and TXT (Trusted Execution Technology), which attests the authenticity of the used platform and operating system. A very important thing to consider here is the trust chain, that is, secure runtime can be considered secure if it is provided by a secure boot, and a secure boot has to be performed on trusted hardware. Lack of trust/security in one of the links results in an untrusted/insecure environment.

Secured crash

Along with the verification of booting and starting phases of a virtual machine, securing the crash of the VM should also be discussed. Some points of concern are highlighted below [2]:

- VNF components: (in general VMs that are running on a hypervisor)
  The hypervisor should ensure that unauthorized entities may not have access to any file references, hardware pass-through devices, or memory that was left by a crashed VM. The same applies to a crashed application inside a VM. In this case, the hypervisor should assure that the crash of the application will not make any authorization changes. This task is harder to perform by the hypervisor because it could be unaware of the application failure.
- References on remote devices to a crashed VM machine:
  Hypervisor should be able to ensure that a newly launched VM will not adopt addresses that were recently used by a crashed machine. Otherwise, the new VM would be able to exploit privileges of the crashed VM and thus grant access to unauthorized parties. This could be mitigated by mandatory authentications performed before requesting certain resources.
- Local and remote storage resources attached to a VM:
  It is impossible for the hypervisor to determine autonomously which part of an attached storage should be wiped after a VM crash because some of those resources could be shared or will be reused in the future. Therefore, such information should be defined manually.
- Swap storage attached to a VM:
  If a VM uses swap storage it should be marked for hypervisor as "swap". In this case, the hypervisor will know that it should be wiped in case of a crash.

Performance and Isolation

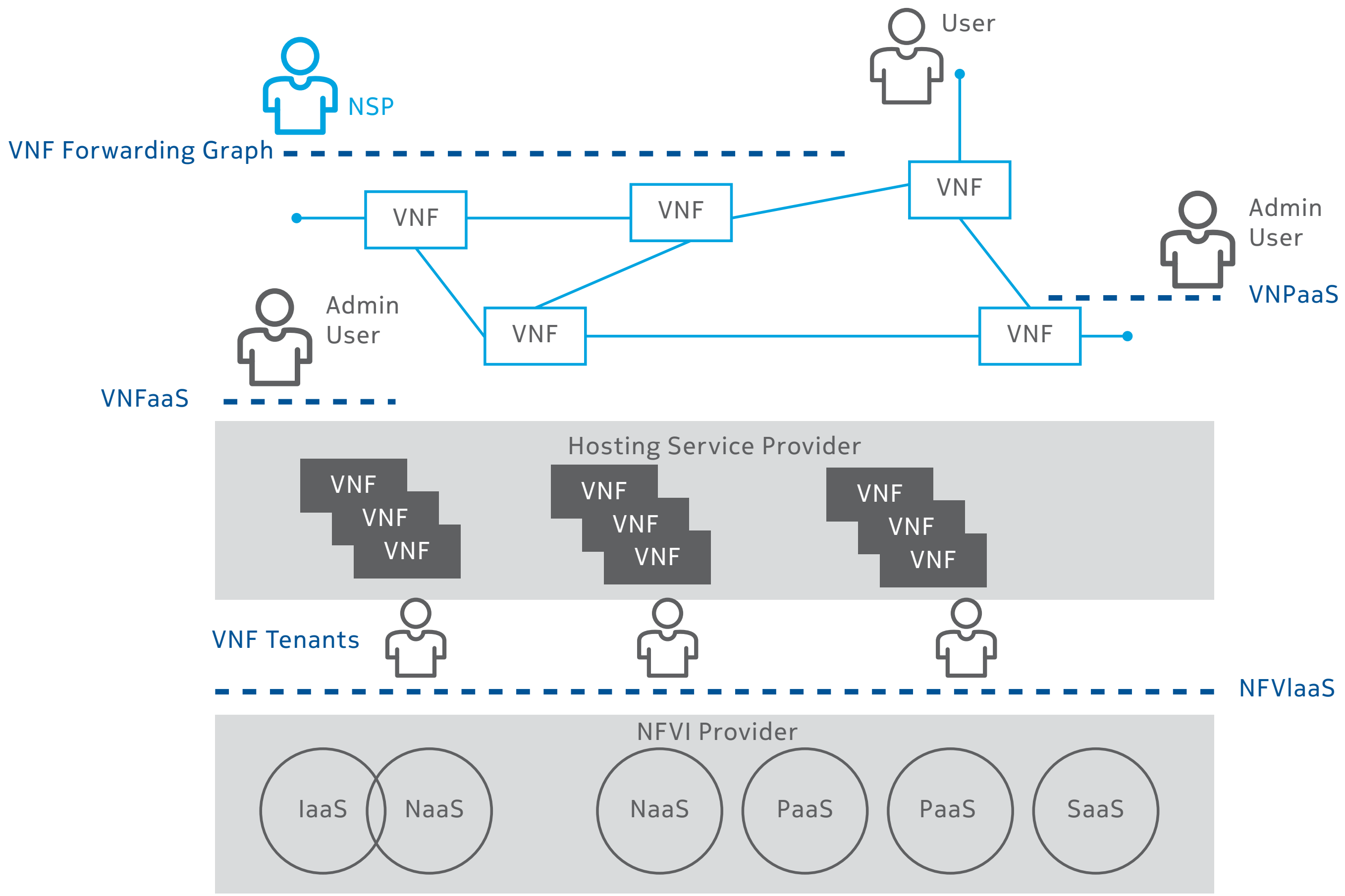
Another issue related to virtualized infrastructure is an attack on the resources that are used by a VNF. There is one simple rule: all the resources of the VNF (except explicitly shared resources) must be separated from other VNFs. However, the weakest point is the virtualization host because if an attacker gains access to it, he will be able to crash the system, degrade performance, or even wipe all the VNFs.

AAA (Authentication, Authorization, and Accounting)

The next challenge involves user/tenant Authentication, Authorization, and Accounting. Figure 4 presents an exemplary scenario.

Users or admins may need access to VNFs that are separated in different clouds/VLANs (horizontal access). At the same time, they may use or manage entities in different layers (VNFs, hypervisors), hence vertical access is also required. This scenario causes a complex roles-and-privileges matrix, which is prone to error. There is no simple solution to this issue. An account manager needs to grant access to users and tenants very carefully, because granting access unreasonably can be dangerous. It should be remembered that, in terms of security, it is always better to grant less access than give more access than needed.

Figure 4 Example of multi-layered identities [2].



Time synchronization

The next concern that should be taken into account is related to time synchronization. The date & time settings are usually synchronized by hypervisor, which is the single source of time for all VNFs [2]. Therefore, manipulation of the time settings on the hypervisor affects all VNFs. Nonetheless, if an attacker gets unauthorized access to it he will be able to perform much more severe attacks against the cloud ecosystem than simple time manipulation.

Private keys distribution and storage

When discussing security, concerns related to private keys distribution cannot be omitted. Ideally, keys, a system root password, and other sensitive data should not be held within the system image. They should be injected during the first VNF boot. Every VNF should have its own key pair (those running in a high availability cluster can share one key pair). In this way, the situation where one compromised machine compromises others can be avoided.

Back-doors via testing interfaces

Another issue affecting most virtualized environments pertains to testing and troubleshooting activities that are usually performed remotely. The common pitfall here is leaving the testing/debug interfaces open after an environment is put in production. They are often accidentally left open but, surprisingly, sometimes they are intentionally left open for easier troubleshooting by a support staff. It is a very risky practice because "security by obscurity" is a commonly criticized approach as it provides only a false confidence that the system is to some extent secured.

5. Conclusion

Virtualization is a powerful technology and, when designed properly, it can bring meaningful benefits including improved security. Nevertheless, more power means more responsibility and the deployment of any cloud environment needs special attention. Virtualization has the potential to make network management simpler because of offered abstraction, yet it always makes the network structure more complex by introducing an additional layer in the system architecture. This virtualization layer has specific vulnerabilities, which brings new challenges for cloud network security architects and engineers. Neglecting this fact may cause fatal consequences since the security of any system is as strong as its weakest element.

References

[1] ETSI GS NFV 001, *Network Functions Virtualisation (NFV); Use Cases.*

[2] ETSI GS NFV-SEC 001, *Network Functions Virtualisation (NFV); NFV Security; Problem Statement.*

[3] Operations support system. (2015, January 18). Retrieved June 5, 2015, from http://en.wikipedia.org/wiki/Operations_support_system

[4] Business support system. (2015, June 3). Retrieved June 5, 2015, from http://en.wikipedia.org/wiki/Operations_support_system

[5] ITU-T Recommendation M.3010 (2000), *Principles for a telecommunications management network.*

[6] ETSI GS NFV 002, *Network Functions Virtualisation (NFV); Architectural Framework.*

[7] ETSI GS NFV 003, *Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV.*

[8] ETSI GS NFV-MAN 001, *Network Functions Virtualisation (NFV); Management and Orchestration.*

About the authors

I have received the M.Sc. in ICT from the Wrocław University of Technology, and am now a Ph.D. student at the Institute of Informatics. My professional and scientific interests are focused on multi-layered virtualization of network resources and QoS mechanisms. Today I work in the MBB Security department with Nokia Cloud Security Director, whose purpose is to secure telco cloud environments.

Bartłomiej Dabiński
Security Solution Engineer
MBB Security

I have worked for two years as a Network Specialist and three years as a Network Security Specialist. My interests are related to security in networks, especially social engineering techniques and methods of protecting multi-layered complex environments. Currently I am working in the MBB Security department. My scope of work is related to the integration of Nokia Network Access Guard.

Marcin Otwinowski
Integration and Verification Engineer for Security
MBB Security

5G for Mission Critical Machine Type Communications

Karol Drażyński
Senior Radio Research Engineer
T&I Radio Research

Maciej Januszewski
Senior Radio Research Engineer
T&I Radio Research

1. Trends in mobile communication and the path to 5G

The number of smart phones, tablets, different types of applications, and video-streaming services that we use is growing. This results in unparalleled diversity of ways in which we connect and communicate. The exponential increase in consumer mobile traffic stemming from a growing number of those devices and services is one of the driving forces behind the development of 5G radio technologies. To accommodate the expected 10 000 times higher traffic level in the year 2020 compared to 2010 and also widely available 100 Mbit/s average throughput, both radio access and the core part of mobile networks need to be redesigned. It is however not the only reason, since there is a number of new services and requirements shaping the future of mobile communication. As a matter of fact the vast variety of service requirements is why new system architecture is needed with programmable, software-driven core networks to efficiently address service demand diversity. The new system also needs to be flexible in order to provide a future-proof platform. This means support for newly-identified services and use cases, as well as for services that have not yet emerged but will very likely appear in 10 years from now. New areas where mobile networks are expected to vastly improve with the creation of 5G are: throughput, latency, battery lifetime, as well as energy and cost efficiency. Improving those Key Performance Indicators paves the way towards Internet of Things (IoT) with new services and applications enabling billions of devices to stay connected and interact with each other. IoT (see **Figure 1**) already today and even more in the future will create a big impact in nearly all aspects of human lives, for example: medical and health care, wearable devices, monitoring services, smart cities, and many types of consumer appliances and sensors.

Among other beneficiaries one can list, industry automation, smart grids, logistics, big data analytics, and last but not least, the automotive industry. To enable those services, reliable, effective, and cost efficient radio communication solution is required, which 5G is targeting to provide. Hence, the Machine Type Communication (MTC) is one of the pillars of the new 5G system.

2. Machine Type Communication overview

Machine Type Communication (MTC) is a type of communication enabling machines to exchange information with the cellular mobile network or with other devices via the cellular networks. It is expected to become an important part of 5G system enabling the full utilization and exploration of IoT services. With predictions that up to 25 billion devices (according to Cisco's VNI Forecast [1]) will be connected to the Internet by the year 2020, mobile networks must ensure that they will be able to satisfy this connectivity demand. Since there will be much more devices than humans connected to the network, MTC type of connections will have a major role in mobile networks' traffic load. This means a shift from traditional mobile networks assumptions in which the human end user is the main beneficiary of voice and data services towards a system designed to also benefit machines and their automated exchange of information. Since machines can

Figure 1 Applications of wireless connectivity in 5G-IoT era.

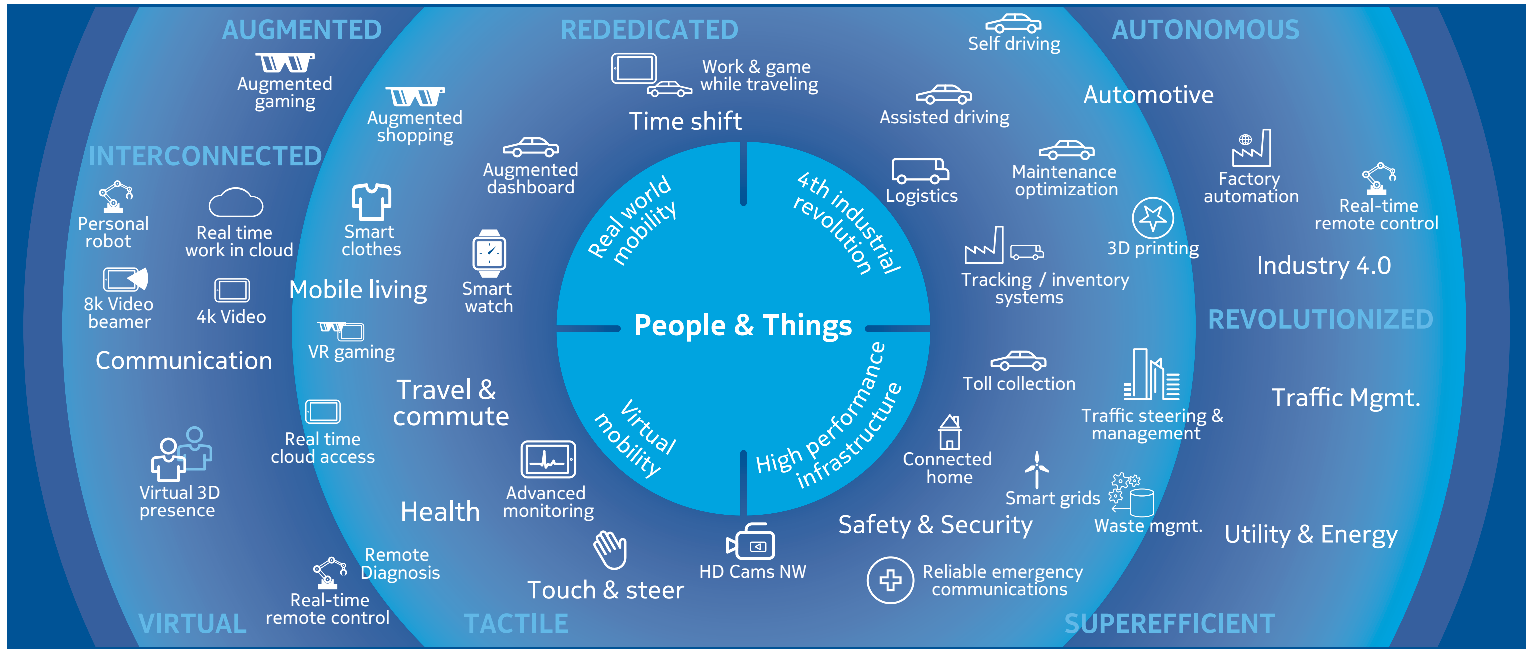
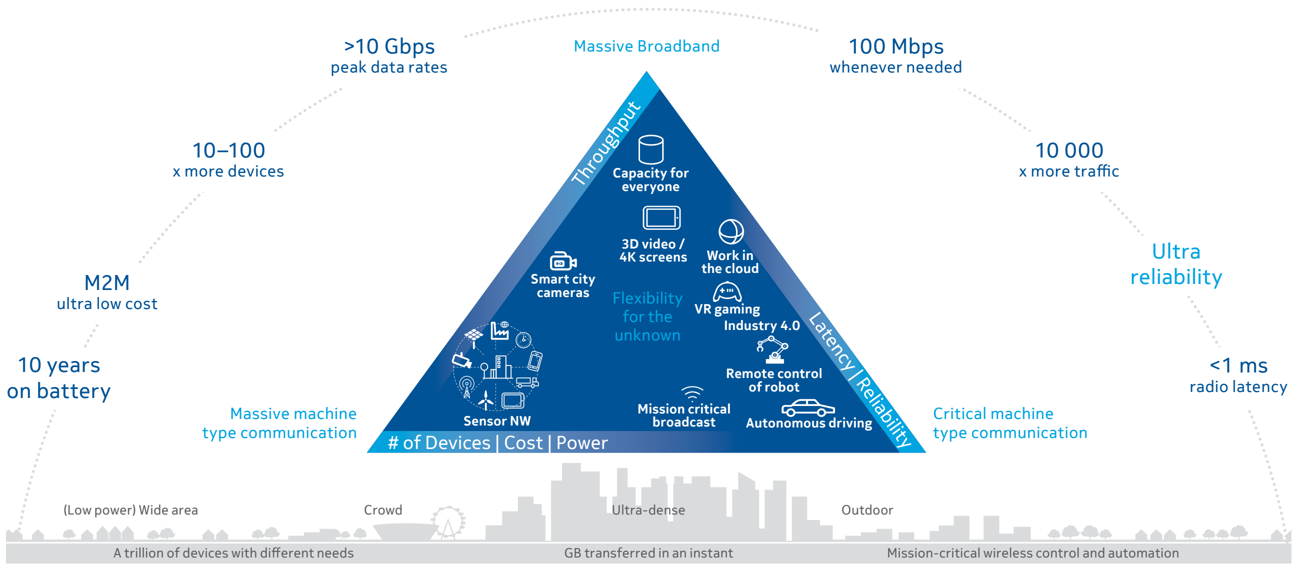


Figure 2 Diversity of services, use cases, and requirements for 5G.



process information much faster than humans, the communication latency level should match the machines processing times. Hence, the expected communication latency has to drop significantly. Also, cellular direct device-to-device communication shall be possible for peer-to-peer data exchange. Examples of such IoT services are self-driving cars, smart homes, or industry automation with inter-connected machines. Certainly, this requires many changes in basic network assumptions in terms of device addressing, security procedures, and service differentiation. Due to the plethora of IoT devices and their applications the resulting traffic has a broad scope of extreme requirements, often requiring advanced QoS management in the radio network. After all, a 5G network will have to accommodate all types of Mobile Broadband traffic and not only MTC traffic. All these possibilities will be ultimately enabled with one 5G radio access and hence there will be a strong need to manage the resulting traffic types efficiently and according to their relevant requirements.

3. Applications of MTC

The aforementioned diversity of applications and service types will also result in different types of connections required for those devices. For instance, we can envision the following applications for 5G MTC [6]:

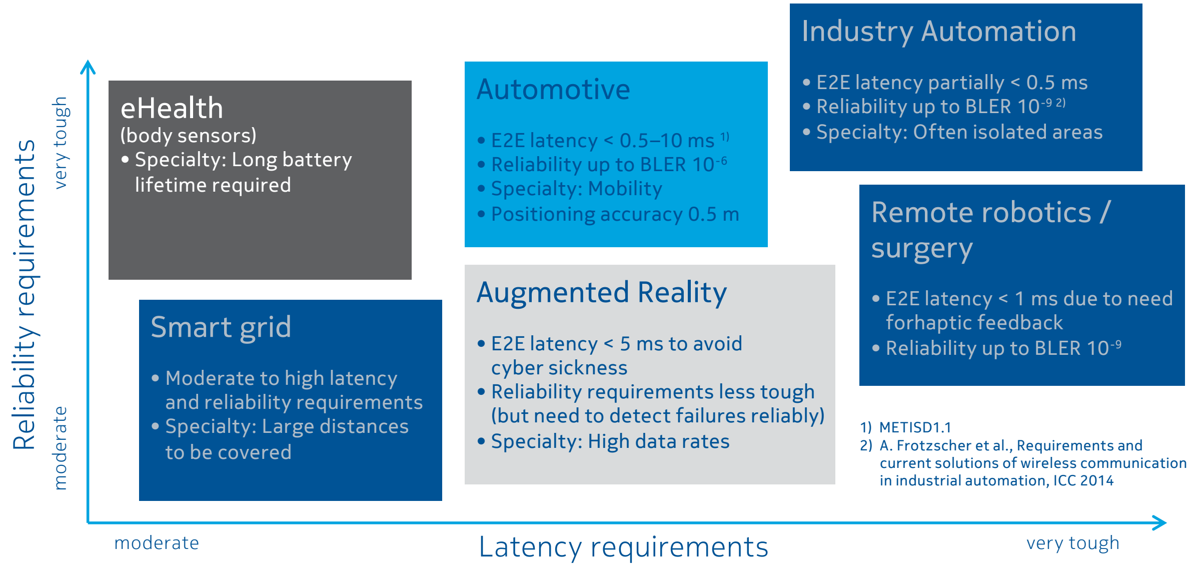
- Autonomous driving
  Cars will be able to drive autonomously, in real-time obtaining information on road and traffic conditions, and exchanging information with other road users and network infrastructure to improve road safety.
- Industry automation
  With interconnected ultra-reliable communication robots and other manufacturing devices, a fully remote plant automation scheme can be provided.
- Smart city
  Ultra-low-power-consumption sensors (wearable or standalone) scattered around the city can enable the exchange of data and provide access to short-range IoT.

The differences between applications and the resulting connection requirements stem from the fact that each of them can have a different set of requirements regarding latency, reliability, availability, and throughput. Other parameters, which need to be considered to differentiate connection types, could be for example traffic type, authentication, or subscription and pricing differentiation.

A common division of MTC is with regard to the number of devices and to the required reliability and latency.

Figure 2 depicts the triangle of services envisioned for 5G. Clearly, the main characteristic of 5G system is that it will have to be much more scalable compared to today's cellular systems. The applications range from those designed for Gigabyte throughputs to the ones requiring ultra-low latency. Even inside the MTC domain the requirements are extremely diverse. The bottom-left corner of the requirements triangle shown in **Figure 2** is often referred to as Massive MTC or the IoT. Here the main challenge is supporting communication of all sorts of network-access-capable devices. These devices are expected to be deployed with a density reaching up to 3 million devices per square kilometer, hence their cost and energy consumption create another challenge so be addressed. Potentially, those devices will not require very high data throughputs, their access to the network is expected to be rather periodic or intermittent with small amounts of data per transfer. The omnipresent wireless sensors and radio tags will not require very high throughputs but should be cheap and very energy-efficient, so that they can operate for at least 10 years on a single battery or even using energy harvesting techniques. The bottom-right corner of this triangle refers to so called Mission Critical Communication. These are the services that require either ultrahigh reliability, very low latency or, in the most challenging case, the combination of both.

Figure 3 Mission Critical Communication use cases.



Reliability requirements
very tough
moderate
eHealth
(body sensors)
• Specialty: Long battery lifetime required
Smart grid
• Moderate to high latency and reliability requirements
• Specialty: Large distances to be covered
Automotive
• E2E latency < 0.5–10 ms 1)
• Reliability up to BLER 10-6
• Specialty: Mobility
• Positioning accuracy 0.5 m
Augmented Reality
• E2E latency < 5 ms to avoid cyber sickness
• Reliability requirements less tough (but need to detect failures reliably)
• Specialty: High data rates
Industry Automation
• E2E latency partially < 0.5 ms
• Reliability up to BLER 10-9 2)
• Specialty: Often isolated areas
Remote robotics / surgery
• E2E latency < 1 ms due to need forhaptic feedback
• Reliability up to BLER 10-9
moderate
Latency requirements
very tough

1) METISD1.1
2) A. Frotzscher et al., Requirements and current solutions of wireless communication in industrial automation, ICC 2014

Figure 3 presents the Mission Critical use cases sorted according to their latency and reliability requirements. As illustrated, the Industry Automation and Remote robotics use cases put the most stringent requirements on the system. High-precision robots need to communicate with ultra-high reliability and latency below 1 millisecond [5]. The eHealth application area is expected to have moderate end-to-end delay needs but has to remain very reliable. Similarly to Smart Grid use cases where the information will have to be transmitted quickly over very long distances for example to prevent energy network failures. Automotive use case remains one of the most interesting in this group. Apart from critical latency and reliability requirements, which cannot be met with the existing wireless technologies, vehicle-to-vehicle communications will also include considerable device and traffic density, as well as very high mobility, hence challenging radio conditions.

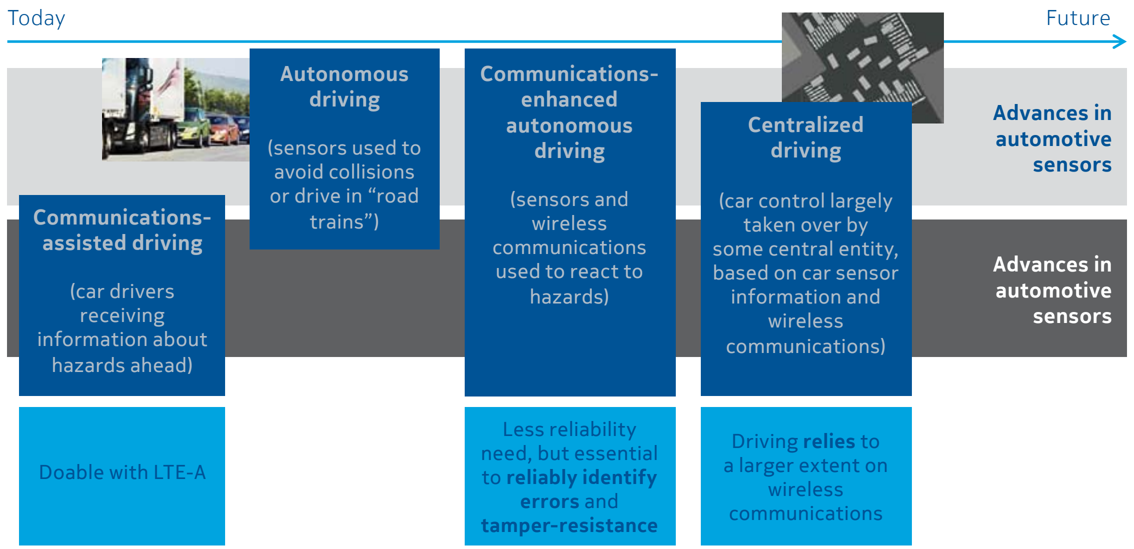
4. Mission Critical vehicle-to-vehicle communication

Anually, 40 000 people die and 1.7 million are injured in result of traffic accidents in Europe alone [2]. Apart from the casualties, increasing traffic leads to jams, pollution, higher fuel consumption, and increased transition time. The heavily-researched Cooperative Intelligent Traffic System (C-ITS) [3] can address these problem by actively warning drivers about road hazards or automatically intervening in dangerous situations. C-ITS could also optimize traffic load and thus reduce traffic jams and fuel consumption. Apart from direct vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication, cooperation with vulnerable road users (VRU), e.g. pedestrians, cyclists, should be possible. It is expected that the personal devices (e.g. smartphones) that are carried by the vulnerable users could also play an active part in the C-ITS.

Naturally, autonomously driving cars will also rely to great extent on onboard sensors. The V2V and V2I communication will only play a supplementary role as long as this communication is not ultra-reliable. However, even in the initial stages of autonomous driving, the vehicle-to-vehicle communication can be extremely useful due to number of reasons:

- Sensors embedded in cars cannot "see" over corners and other cars.
- Sensors' operation can be compromised in poor weather conditions.
- Sensors introduce a considerable reaction delay: currently 100–200 ms.
- Receiving the same information over multiple channels (sensors and radio) increases reliability.

Figure 4 Phases of autonomous driving.


Today
Future
Communications-assisted driving
(car drivers receiving information about hazards ahead)
Autonomous driving
(sensors used to avoid collisions or drive in "road trains")
Communications-enhanced autonomous driving
(sensors and wireless communications used to react to hazards)
Centralized driving
(car control largely taken over by some central entity, based on car sensor information and wireless communications)
Advances in automotive sensors
Advances in automotive sensors
Doable with LTE-A
Less reliability need, but essential to reliably identify errors and tamper-resistance
Driving relies to a larger extent on wireless communications

Improvements in both wireless communication and sensors technology are needed to build a truly 100% reliable C-ITS system. Most likely the autonomous driving will be deployed in phases as depicted in Figure 4. However, the early forms of autonomous driving can already be served with 4G LTE-A technology in combination with the IEEE 802.11p standard.

In most phases, C-ITS system will depend on timely and reliable exchange of information via radio communications. Data can be sent either periodically (10–100 Hz) or on the event-triggered basis. The data packet size will be moderate for most use cases, as the messages will contain the information about vehicle id, speed, direction, position, and so on [4]. Nevertheless, there are also some applications that will come along with higher throughput demand. One example of such an application is the see-through technology, which projects a video from the front of the car on its back allowing drivers behind to "see through" that car.

5. 5G solutions that can address Mission Critical Communication needs

Signal diversity is the key to ultra-reliable communication. In a very dynamic multipath vehicular communication channel, the radio conditions can change very rapidly. Different forms of diversity can be combined to overcome a fade in channel quality and ensure successful transmission of the message:

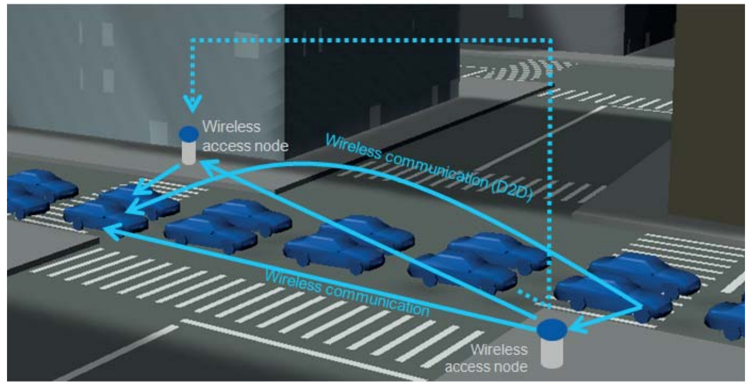
- Time diversity
  Some data is transmitted many times or a redundant error correcting code is added to combat instantaneous error bursts in a channel. In case of Mission Critical communication requiring ultra-low latency the time margin for performing this type of diversity may be limited.
- Frequency diversity
  The data is sent using multiple frequency channels or spread over a wide spectrum to combat frequency-selective fading.
- Space diversity
  Using multiple transmitting/receiving antennas to transmit the signal over multiple paths.

Even though the diversity techniques are well known in wireless technology, the Mission Critical Communication require support for even more diversity, and therefore induce the formation of novel radio solutions and network deployments. It is considered that messages will not only have to be transmitted over multiple antennas but also over multiple completely independent and redundant communication links.

Figure 5 shows how diversity, and thus reliability, can be exploited in a road traffic scenario improving safety while using multiple links to convey the same message. One of the fundamental techniques used here is direct device-to-device (D2D) communication which means that devices in close proximity can transmit to each other directly and offload the Base Stations (BS). However, in case of V2V communications the BS or dedicated Road Side Unit (RSU) can be used in parallel to improve the reliability.

There are also numerous enhancements that are considered to be deployed in 5G to help in dealing with Mission Critical Communications. One example of those is a new design of the Physical Layer. Shorter transmission time interval will reduce the end-to-end delay in the network. New frequency bands expected to be available for 5G will open much more spectrum which will considerably improve the transmission quality. Advances in receiver technology will reflect in better interference cancellation, while access design and coding technique enhancements will optimize the transmission for small data packets and ensure faster decoding.

Figure 5 Vehicles-to-vehicle communication over multiple links.



6. Vision for MTC in 2020 and beyond

The development of 5G technology will provide, an efficient way to exchange information between machines. Major differences compared to human type communication are related to speed of information exchange, latency, as well as throughputs. Machines can simply process information faster than humans and so their communication means have to match that. Due to the fact that support for MTC will be embedded from the beginning into 5G it will enable the deployment and utilization of Internet of Things (IoT). Billions of devices will be able to communicate and exchange information with other machines to help people fulfill their everyday tasks. One of the major breakthroughs with this technology will be the introduction of "intelligent" vehicles, being able to communicate with each other and the rest of the road users. Over the years the vehicles will gradually transform into fully autonomous, self-driven cars. To enable this type of applications, 5G needs to support MTC with ultra-reliable and latency-critical radio links. This challenge has so far never been achieved and only a new system, designed from the scratch to fulfill this task can enable such solutions.

It remains to be yet discovered which new applications and services will emerge with the introduction of 5G and Mission-critical MTC. For sure it will transform our daily interactions with machines and cars to guarantee safer and more comfortable journeys.

References

[1] Cisco VNI Forecast Highlights, http://www.cisco.com/web/solutions/sp/vni/vni_forecast_highlights/index.html
[2] http://ec.europa.eu/transport/road_safety/specialist/statistics/index_en.htm
[3] ETSI Cooperative ITS, http://www.etsi.org/index.php/technologies-clusters/technologies/intelligent-transport/cooperative-its
[4] ETSI TS 102 731 v1.1.1, ITS Security Services and Architecture.
[5] Nokia 5G White Paper, http://networks.nokia.com/file/28771/5g-white-paper
[6] Nokia 5G Requirements White Paper, http://info.networks.nokia.com/5G_Requirements_wp.html

About the authors

I have graduated from Poznań University of Technology in Electronics and Telecommunications. I am currently working in Technology and Innovation (T&I) department, where I am developing concepts for the 5G system with focus on Machine Type Communication with Mission Critical deployments

Karol Drażyński
Senior Radio Research Engineer
T&I Radio Research

I work in Wrocław Technology and Innovation (T&I) department of Nokia Networks. I lead the research on Mission-critical Machine-type Communication for 5G. I'm also leading the Innovation Board where all employees from Nokia Wrocław can submit their innovative ideas that improve our products and workplace.

Maciej Januszewski
Senior Radio Research Engineer
T&I Radio Research

Big-data-driven Telco Market

Sławomir Andrzejewski
Specialist, Network Engineering
MBB Customer Support

Ireneusz Jabłoński
Specialist for Data Mining
and Machine Learning
MBB Customer Support

John Torregoza
Specialist, Network Engineering
MBB Customer Support

Krzysztof Waściński
Radio Concept Owner
MBB Customer Support

1. Looking through a new microscope of Big Data

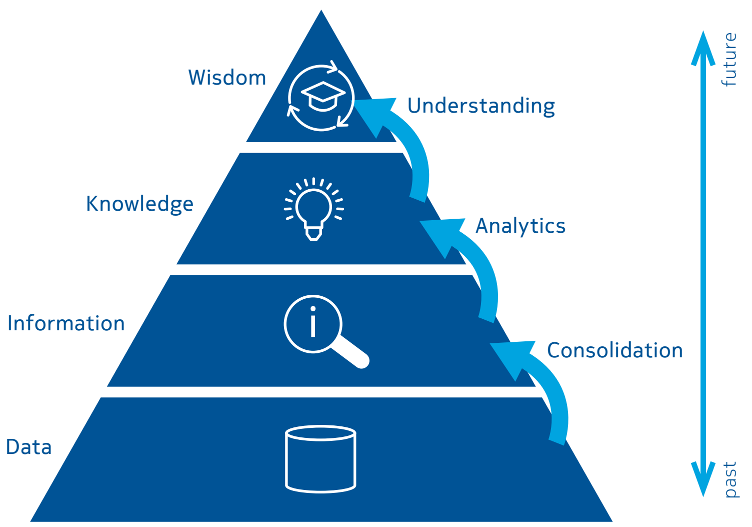
Mankind has come a long way from the Stone Age era to the world we are living in today – the Big Data (BD) era. This new mind-blowing universe is expanding incredibly fast – 90% of data that ever existed have been created in last two years [1, 2]. The amount of data available now is huge, but even more fascinating is the fact that there is a plenty of undiscovered connections in this space. Therefore questions raise: How to handle that data? How to get meaningful results? As a starting point on the path to wisdom, data is the result of an observation (measurement) and it is generated in the interaction between people, machines, applications, and the combinations of these. There is no other way clarify what is going on in the modern world. A bridge between data and wisdom need to be found (**Figure 1**). Here, data is the most basic level, information adds context, knowledge adds to how to use it, and wisdom adds to when and why to use it.

But in fact, there is a huge amount of people, machines, applications, and other elements in the world, which imply many possible differences in combinations spanned between them. Moreover, the ecosystem is complex and it keeps evolving.

Now, let's have a closer look on the four Vs: Volume, Variety, Velocity and Veracity (see **Figure 2**).

Volume refers to the magnitude of data (see **Figure 3**). Definitions of big data volumes are relative and vary by factors, such as time and data type. Some think that what is considered a big data today may not meet the threshold for big data in the future, because storage capacities will increase, allowing even bigger data sets to be captured.

Figure 1 'Big Data' concept and D-I-K-W hierarchy in cognition process.



Variety refers to the structural heterogeneity in a dataset. Technological advances allow companies to use various types of structured, semi-structured, and unstructured data. Structured data, which constitutes only ~5% of all existing data [4], refers to the tabular data found in spreadsheets or relational databases. Text, images, audio, and video are examples of unstructured data, which sometimes lack the structural organization required by machines for analysis.

Figure 2 Four Vs features identified for Big Data [3].

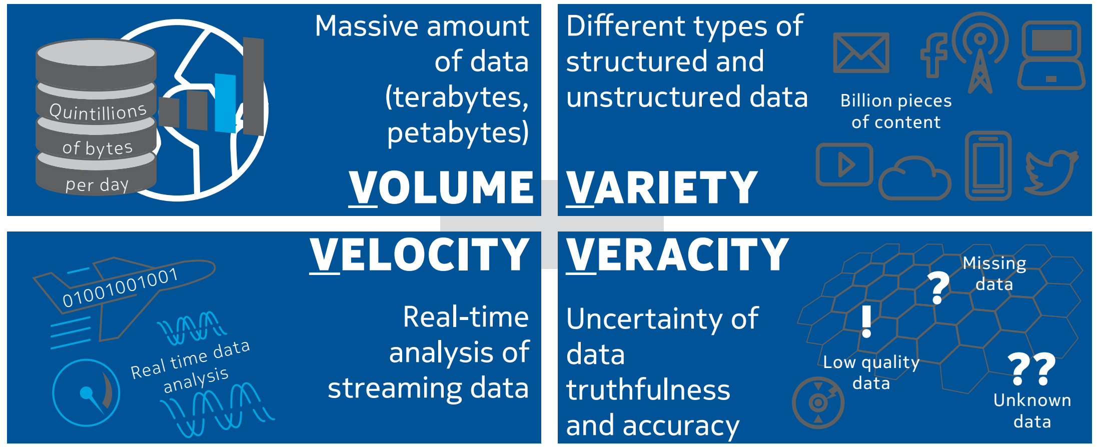
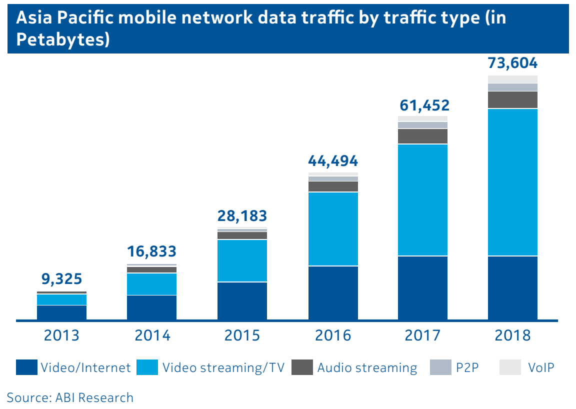


Figure 3 Storage terms.



Source: ABI Research

Prefix	Symbol	10^n
kilo	k	10^3
mega	M	10^6
giga	G	10^9
tera	T	10^{12}
peta	P	10^{15}
exa	E	10^{18}
zetta	X	10^{21}
yotta	Y	10^{24}

1 Petabyte
1,000 Terabytes or
250,000 DVDs

480 Terabytes
A digital library of all of the world's catalogued books in all languages

100 Petabytes
The amount of data produced in a single minute by the new particle collider at CERN

1 Exabyte
1,000 Petabytes or
250 million DVDs

5 Exabytes
A text transcript of all words ever spoken †

100 Exabytes
A video recording of all the meetings that took place last year across the world

400 Exabytes
The amount of data that will cross the Internet in 2012 alone

1 Zettabyte
1,000 Exabytes or
250 billion DVDs

1 Zettabyte
The amount of data that has traversed the Internet since its creation

300 Zettabytes
The amount of visual information conveyed from the eyes to the brain of the entire human race in a single year ‡

1 Yottabyte
1,000 Zettabytes or
250 trillion DVDs

20 Yottabytes
A holographic snapshot of the earth's surface

Velocity refers to the rate at which data are generated and speed at which it should be analyzed and acted upon. The proliferation of digital devices such as smartphones and sensors has led to an unprecedented rate of data creation and is driving a growing need for real-time analytics and evidence-based planning.

Veracity reflects the level of uncertainty of the data. In the imperfect world we live in, the data is imperfect too. Usually there are a lot of inconsistencies or missing parts. As a result we might consider and process collected data only with some level of probability (see **Figure 2**).

Bear in mind that your particular problem does not necessarily have to combine all the big Vs to be labeled as a Big Data problem. You might be challenged to deal with a relatively small amount of data in terms of volume, but at the same time with a great complexity. Big Data is more about getting wisdom from data than anything else. Finally, contemporary Big Data studies show that the external data carry much more information about the system than the internal.

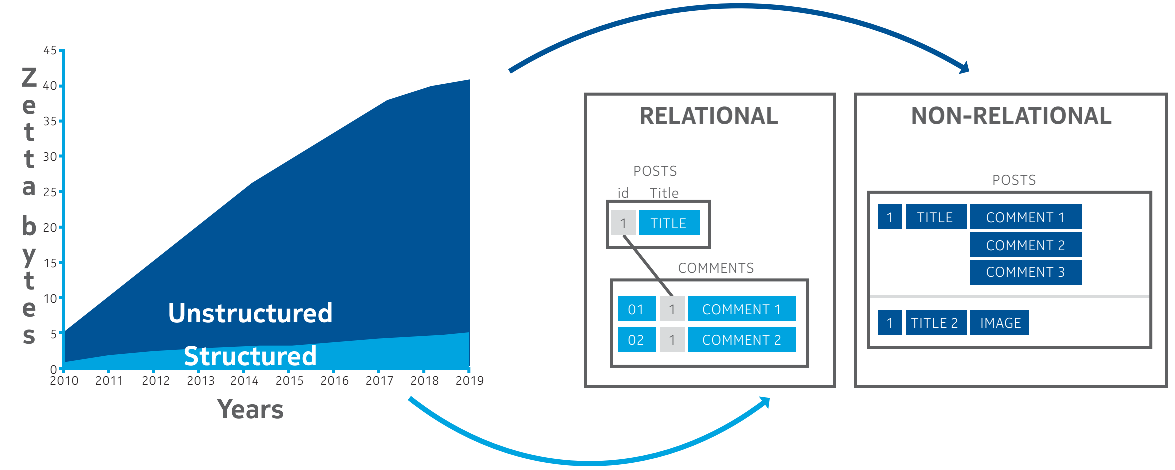
And indeed it is the knowledge and wisdom which determine the final objective for a success story in business. Nokia is looking for the general laws conditioning a continuous progression in modern business, which cannot do well without the (tele)communication layer. Despite the power of the term of 'Big Data', we need to be conscious that data fall into the past, but the future for the telco market is the art of closing the gaps in the D-I-K-W scheme with regards to dealing with (big) data (**Figure 1**). Put simply, currently a frontier of big data expresses itself through extreme information management.

Speaking in both local and general sense, the open question is: How can we manage big datasets in order to find singular information, knowledge, and wisdom, which is the precondition for increasing the satisfaction of customers, and finally growth of the telco business? Big Data techniques and Knowledge Discovery approach might help in crucial fields for business in the following ways:

- Deliver smarter services that generate new sources of revenue.
- Transform operations to achieve business and services excellence.
- Build smarter networks to drive consistent, high-quality customer experience.

Remarkable progress has been achieved, but still there are many challenges which should be addressed:

Figure 4 Estimated growth in structured vs. unstructured data and relational vs. non-relational model of management in database system.



- Identifying the 'big' questions significant for progression of telco market.
- Finding concise responses for the 'big' questions using analytic approach,
- Queuing and synchronization of new products, which influences the strategy of the company and profiles the market state/evolution.
- Designing and releasing of new features for Nokia's products.
- Merging the technical and business inputs for establishing balanced growth of the business.

2. Tools landscape for big data

Big data technology includes three main services:

- Data storage
- Data analytics
- Data visualization

In practice, they usually are individual layers (with own methodologies, languages and software tools), which need a significant effort and time to expertise.

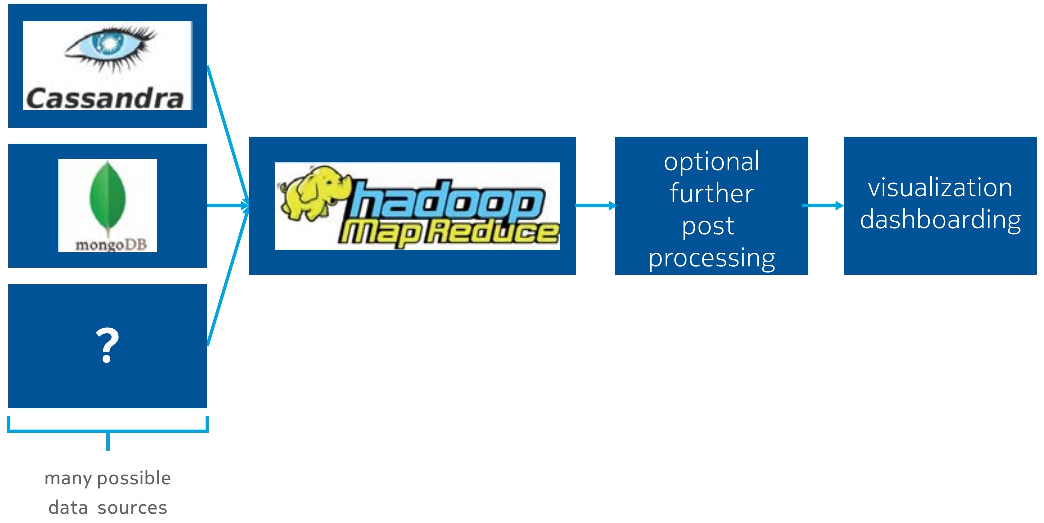
The key issues for each of these services are volume and inherent character (structured, hybrid, unstructured) of data used for inference on complex systems characteristics in Big Data technology. Data features (see **Figure 2**) have triggered BD migration from relational (RDBMS – relational database management system) to non-relational model of management in database system (see **Figure 4**).

Each relational database (DB) is a collection of tables (called relations) consisting of set and relational algebra. For sure excel or .csv files were the starting point in data storage for big data rookies. Simplicity of RDBMS model rules has enabled founding the SQL language (*Structured Query Language*), efficient for algorithmisation of creation, maintaining, and querying of this databases. It should be noted, that still 7 of 10 most popular DB engines use relational rules [5]. Good examples are Oracle or MySQL. But inhomogeneous and dynamic nature of Big Data domain makes data storage a complex task limiting operational workflow in the relational scheme. The cure for this throttle has shown to be a non-relational model of data management (or NoSQL, often interpreted as 'Not only SQL') – see **Figure 4**.

But in contrast to RDBMS, scalable for much of the data processing and business intelligence cases, there is no generalized non-relational scheme used for data storage in BD technology. Now, here, one can easily figure out the demands for BD applications operating with low-latency (real-time or nearly real-time) on huge and inhomogeneous datasets in police systems, navigation systems, telemedical infrastructures, telecommunications networks, etc. Below are ten properties such a system should have [6]:

- scalability
- tiered storage
- self-management
- content availability and accessibility
- analytical and content applications support
- workflow automation support
- legacy applications integration
- enable integration with public, private, and hybrid cloud ecosystems
- self-healing

Figure 5 Exemplary pipeline of big data stream.


Cassandra
mongoDB
?
hadoop MapReduce
optional further post processing
visualization dashboarding
many possible data sources

In fact, each of these characteristics may be more or less relevant, but taken together they provide a vision of how Big Data could be managed in the long term and in an affordable way. Exemplary and popular NoSQL solutions are MongoDB and Cassandra. But, once again referring to the D-I-W-K schema, just having the data is not enough, and processing extensive data might be challenging. The good news is that there are tools and platforms on the market, which are ready to help you with this task. One of the possible paths to be chosen is to use the power of the MapReduce algorithm, which allows the user to benefit from distributed, parallel data processing. The good starting point for understanding how the algorithm works is getting familiar with functions like "mp" and "reduce," widely known from functional programming. Probably the most popular and well known framework related to MapReduce is an open-source project called hadoop (**Figure 5**).

The core of the work of the Nokia Big Data team is focused on designing and implementing BD methods and tools in order to obtain information, knowledge, and wisdom on:

- the temporal and future status of the telco market,
- the optimized leading of Nokia's business and the role of the company in profiling the world telco market.

This is a formalized field of knowledge and skills associated with creative actions, thus the team members are often called data scientists. The data scientist community tends to use a variety of tools, typically across different programming languages. Workflow that involves many different tools requires a lot of context-switching (**Figure 6**).

Figure 6 Diagram of work for data scientist.

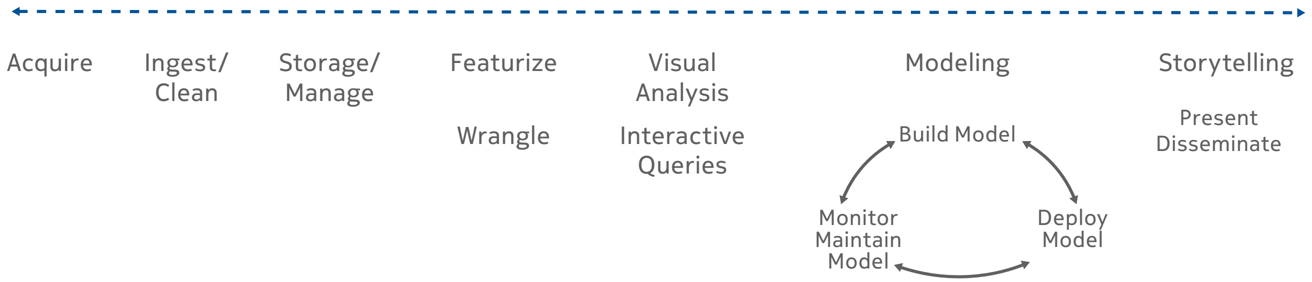
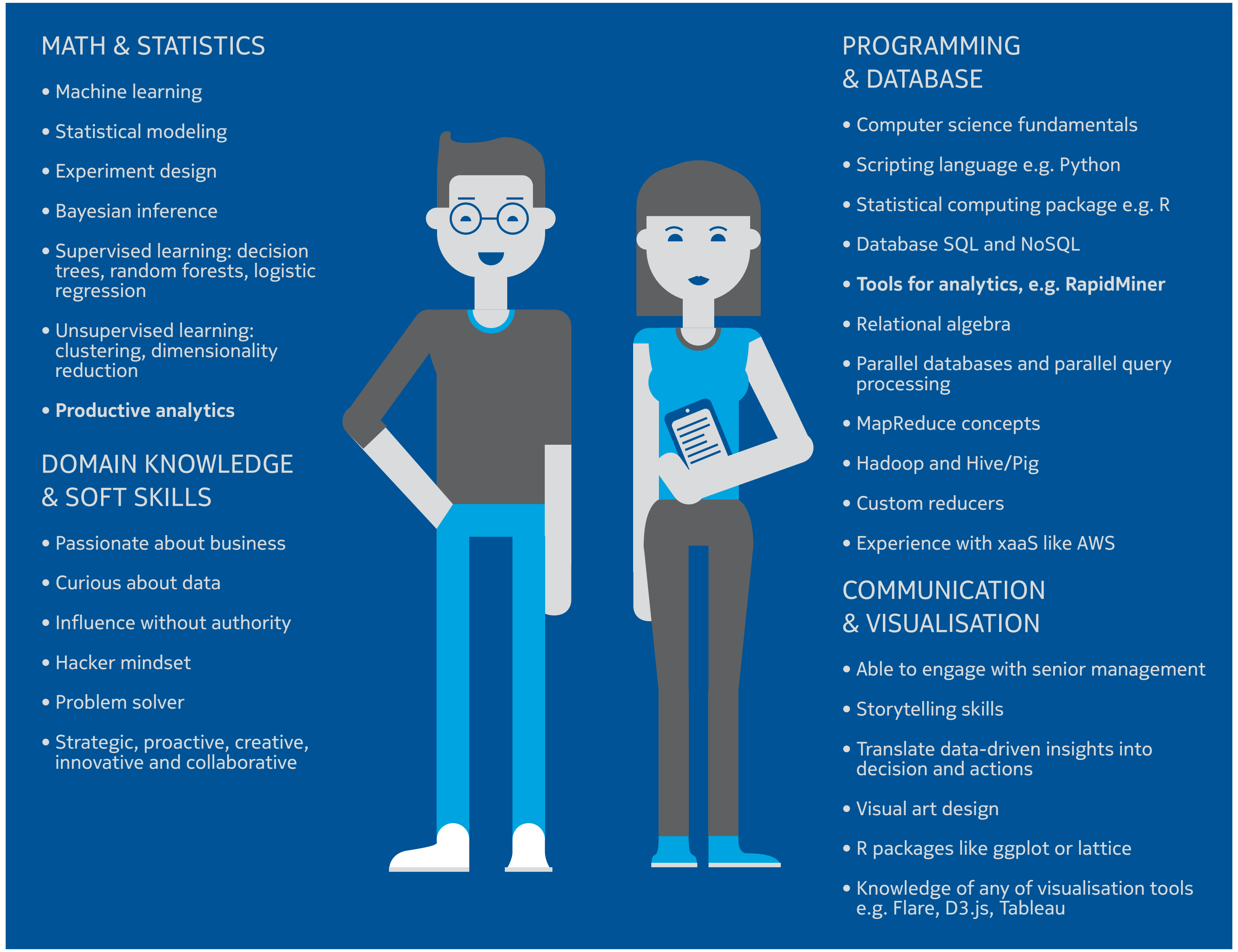


Figure 7 The general profile of a data scientist – a multidisciplinary specialist working at the forefront of modern business.



MATH & STATISTICS

- Machine learning
- Statistical modeling
- Experiment design
- Bayesian inference
- Supervised learning: decision trees, random forests, logistic regression
- Unsupervised learning: clustering, dimensionality reduction
- **Productive analytics**

DOMAIN KNOWLEDGE & SOFT SKILLS

- Passionate about business
- Curious about data
- Influence without authority
- Hacker mindset
- Problem solver
- Strategic, proactive, creative, innovative and collaborative

PROGRAMMING & DATABASE

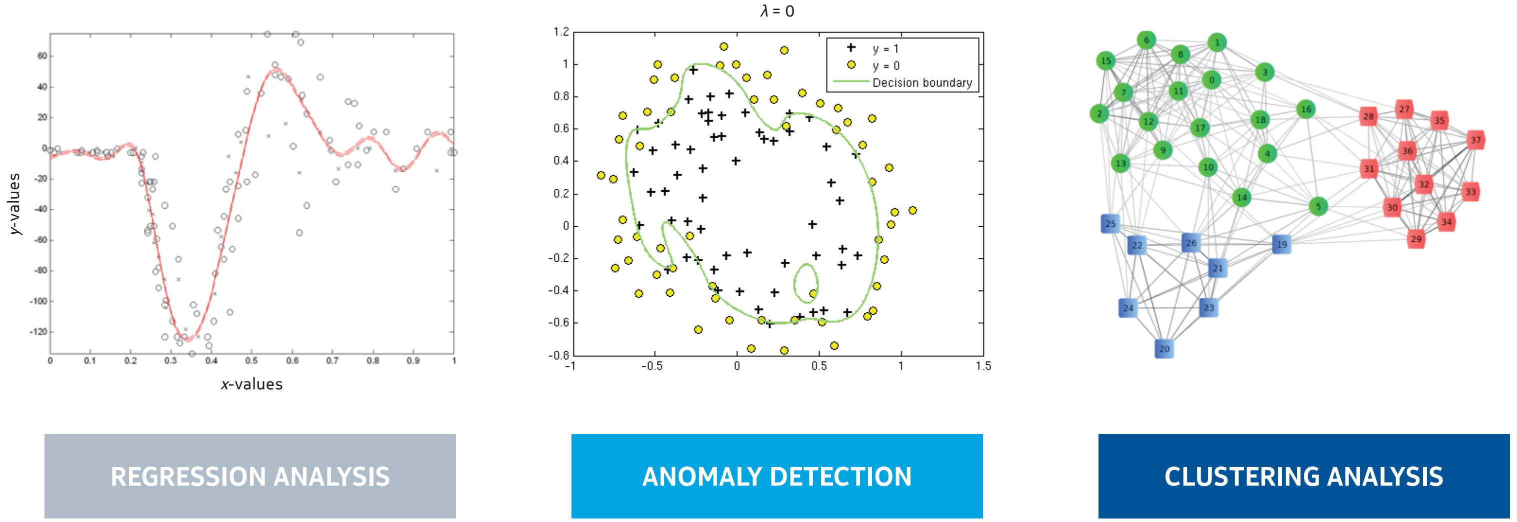
- Computer science fundamentals
- Scripting language e.g. Python
- Statistical computing package e.g. R
- Database SQL and NoSQL
- **Tools for analytics, e.g. RapidMiner**
- Relational algebra
- Parallel databases and parallel query processing
- MapReduce concepts
- Hadoop and Hive/Pig
- Custom reducers
- Experience with xaaS like AWS

COMMUNICATION & VISUALISATION

- Able to engage with senior management
- Storytelling skills
- Translate data-driven insights into decision and actions
- Visual art design
- R packages like ggplot or lattice
- Knowledge of any of visualisation tools e.g. Flare, D3.js, Tableau

Although the workflow of a data scientist seems to be complex [7] and requires interdisciplinary training (**Figure 7**), the same work guarantees job satisfaction with highly useful results for top managers developing business strategies. However do not be intimidated by this complex path dominated by a plethora of methods and programming tools. One can start as a trainee, e.g. with simple examples of statistical analysis or machine learning algorithms (**Figure 8**), and develope competence under our supervision. By becoming a data scientist in Nokia, you earn not only a competitive professional profile, but most of all you join the future of the telco market and beyond. Practicing with various software environments for BD analytics and visualization, you can uncover their usefulness in BD tasks which will define your growing expertise – see the examples in **Table 1**.

Figure 8 Examples of applications of machine learning in Big Data analytics.



3. Big Data use cases

The availability and collection of huge amounts of varied data is driving the conception of different big data use-cases. These use-cases are aimed at acquiring valuable insights from the vast amounts of data in order to propose decisions or actions that would improve the business processes within the company. Telecommunications equipment vendors, like Nokia, are in a good position to take advantage of information collected from their systems. In particular, this information can be used to drive sales cases generation and operation efficiency.

Below is a list of the services expected for release by the telco market:

- service and sales automation
- upsell discovery
- business inteligence information enrichement with e.g. geodata
- customer financial spending forecasting
- network audit automation
- anomaly detection & diagnosis
- accurate capacity planning
- cell-site optimization, through machine learning and correlations etc.
- problems root-cause analysis

In this section, examples of big data use-cases have been shown to report the significance of big data and analytics in the area of telecommunications.

3.1. Correlation studies

- Scope of a use-case:
  One of the main pillars of a well performing network is optimization. Even the best-in-class telecommunications gear needs tuning and proper parameterization adjusted to environment characteristics. That task requires system expertise. All these features define the final network performance Key Performance Indicators (KPIs). The legacy methods used to bring KPIs to their maximum level utilize manual tests and KPI analysis on a rather small scale – within single clusters or even sites. Big Data brings that exercise to the next level by a massive data analysis opportunity. On top of that,new, previously unknown dependencies and correlations can be found. Correlation matrix and decision tree methods are used.
- Inputs for analysis:
  Network performance KPIs (Key Performance Indicators) and parameterization (CM – configuration management data).
- Results of analysis:
  The exemplary matrix (symmetrical) is show in **Figure 9**, visualizing the dependencies between 5 measures. In this example, it can be seen that an increase of users correlates positively with the physical resources usage (PRB) measure. Such matrices may include much more metrics, which speeds up the analysis process.

Figure 9 KPI correlation matrix (please note that the matrix is symmetrical, thus axis labels are in the middle).

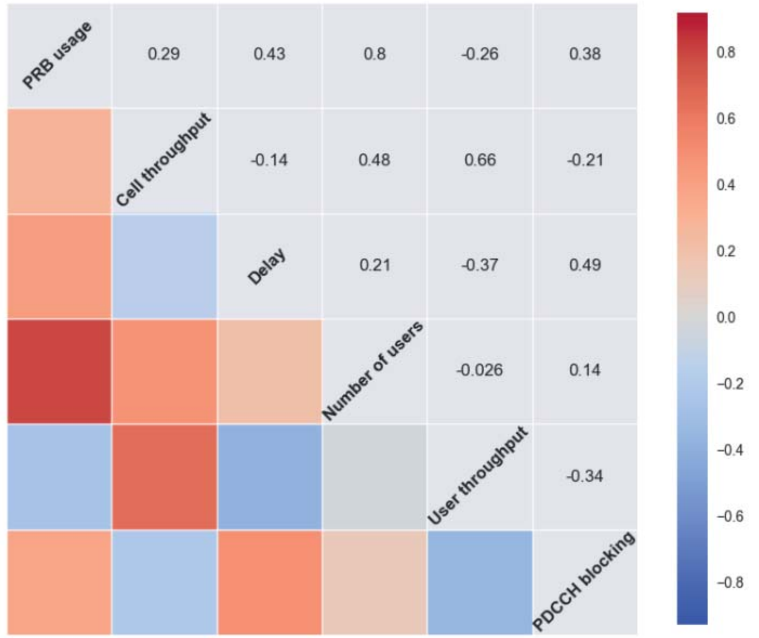
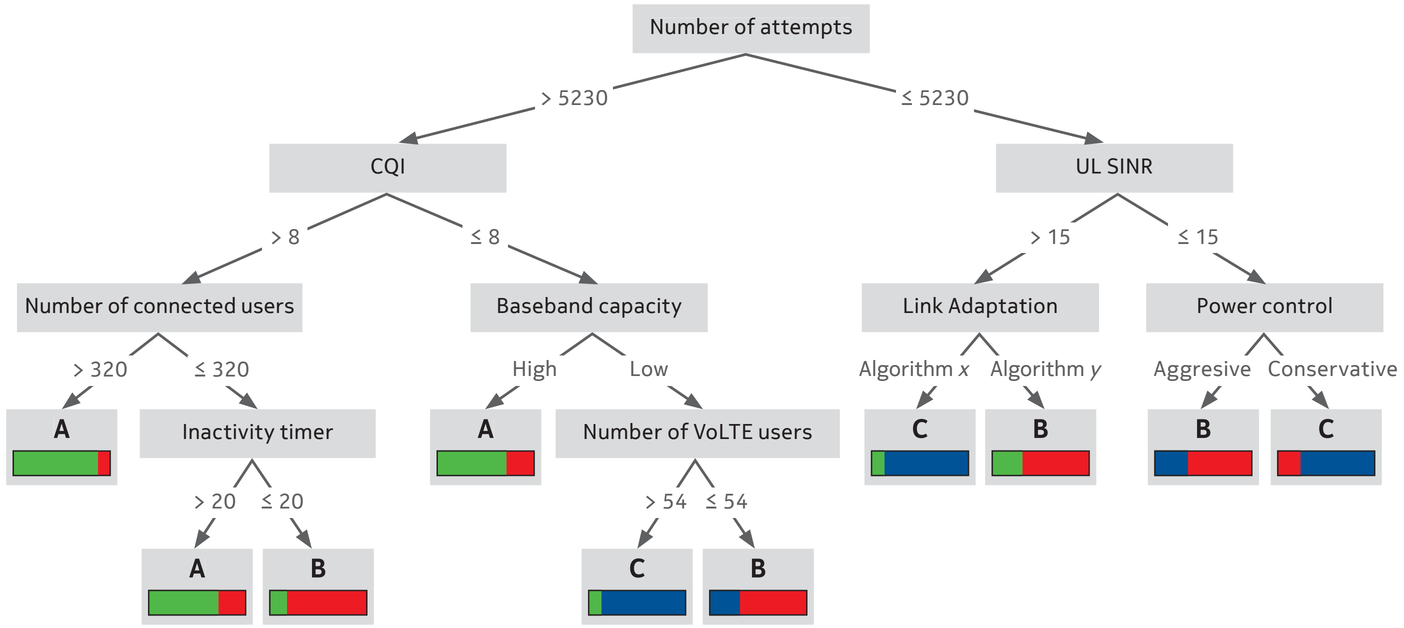


Table 1 The pros and cons of example software tools dedicated to Big Data analytics and visualization.

Name of software tool	Pros	Cons
Matlab	• very well documented • huge number of available functions • good community and support – widely used by engineers and scientists • good IDE (but it is matter of taste)	• not free solution (with toolboxes can be expensive) • requires understanding "matrix-first" concept; might be bit strange at the beginning for programmers switching from other languages
Python	• modules related to data analytics to start with: Pandas (data structures and data analysis), NumPy (scientific computing), MatPlotLib (plots and charts – matlab-like style!), scikit-learn • its general purpose language so other activities/tasks can be solved without jumping out of it (web-frameworks, parsing any kind of files, playing with www/url, creating stand-alone app and many more...) • it's free – most of modules are open source, free to use • huge community, easy to find the answer for your question in web	• it might take some time to choose your favorite modules, set-up your workflow etc. But on the other hand it is fun and because of such fine-tuning exactly according to your needs it might be very effective • documentation is distributed between the modules • sometimes something is missing and/or in experimental state
Rapid Miner	• drag-and-drop interface – design for non-programmers • designed for data science/analytics – build in a wide range of statistical models • free for starting package	• quite expensive for versions higher than starter • more advanced functionalities are accessible in commercial version
R	• very popular amongst scientists and data analysts working in industry • Fortran, C/C++, and Python wrappers are in place • external packages are almost daily increasing, most of them based on published up-to-date books and peer-review articles	• everything you want to do is a command line, minimal GUI • memory management problems, especially when you are working with big datasets

Figure 10 Decision tree.



The inclusion of further, less dynamic, measures, like parameters, encourages the use of decision tree analysis (**Figure 10**). The example shown below the classification model which splits the cells into three different traffic classes (A, B, C) accordingly to their load and features. This representation of data modeling has an advantage over other approaches by being meaningful and easy to interpret.

3.2. Network density analysis

- Scope of a use-case:
  One way to differentiate and characterize customer networks is through the analysis of how densely customers deploy their networks. Depending on the density of the customer's network, there are different network behaviors and recommendations. For instance, for less dense networks, network densification can be given depending on the state of congestion or coverage of the sparse network. On the other hand, for dense networks interference mitigation solutions can be a more helpful recommendation.
- Inputs for analysis:
  In this big data use-case, the geographical coordinates of the eNodeBs are used. These geographical coordinates are either taken from site configuration information or from interpolated coordinates collected and calculated by the HERE application. Based on these coordinates, the distance to the nearest neighbor is determined via Matlab scripting. This distance is then used to characterize the eNodeB.
- Results of the analysis:
  Using the nearest neighbor distance attributes, each eNodeB is classified into different clutter types. For this procedure, a self-organizing map (SOM) algorithm from Rapidminer is used. The SOM algorithm is a neural network based algorithm which maps similar data samples together and differing samples farther from each other. The effect is the formation of clusters which follow the distribution of the samples.

Figure 11 shows a summary of the result of an analysis between two operators. SOM analysis resulted in four clutter types with the average cell range for each clutter type per operator illustrated in **Figure 11**. It can be seen that Operator B has a significantly dense deployment compared to Operator A.

Figure 12 shows another view of the difference between deployments of Operator A and B. This plot shows the cumulative distribution function (CDF) of the operators's deployment. From the figure it can be seen that 50% of Operator A's eNodeBs have a cell range below 1.287 kilometers. In contrast, 50% of Operator B's eNodeBs have a cell range below 1.816 kilometers.

4. Sumary

Big data is an amalgamation of many trends: extreme data growth, a greater importance of external data over internal data, and the shift in computing business models. Big data is about redefining what data

Figure 11 Average cell range.

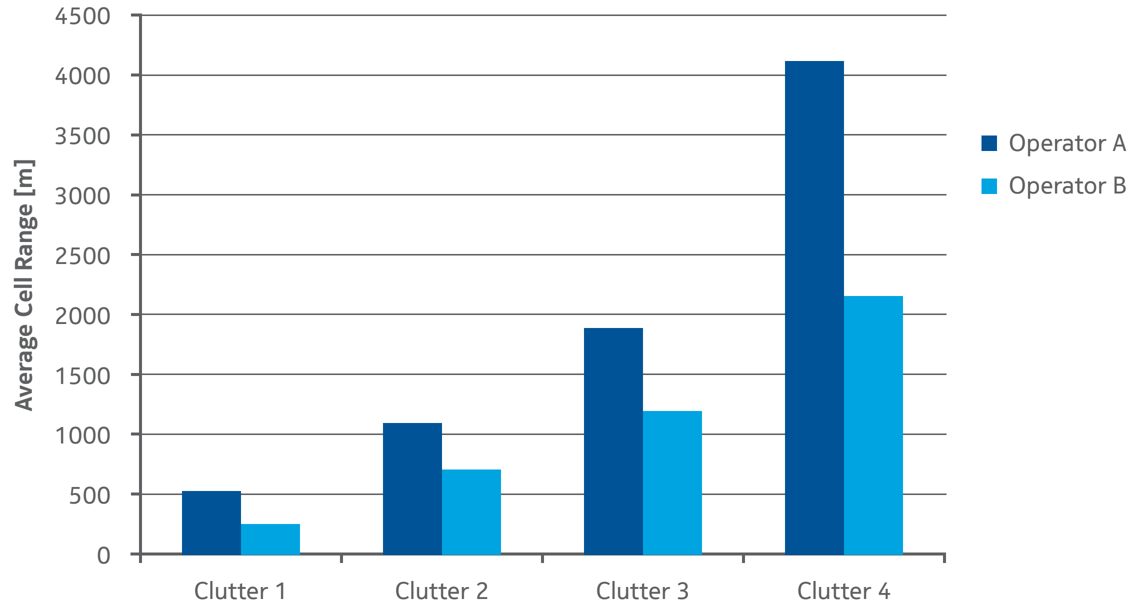
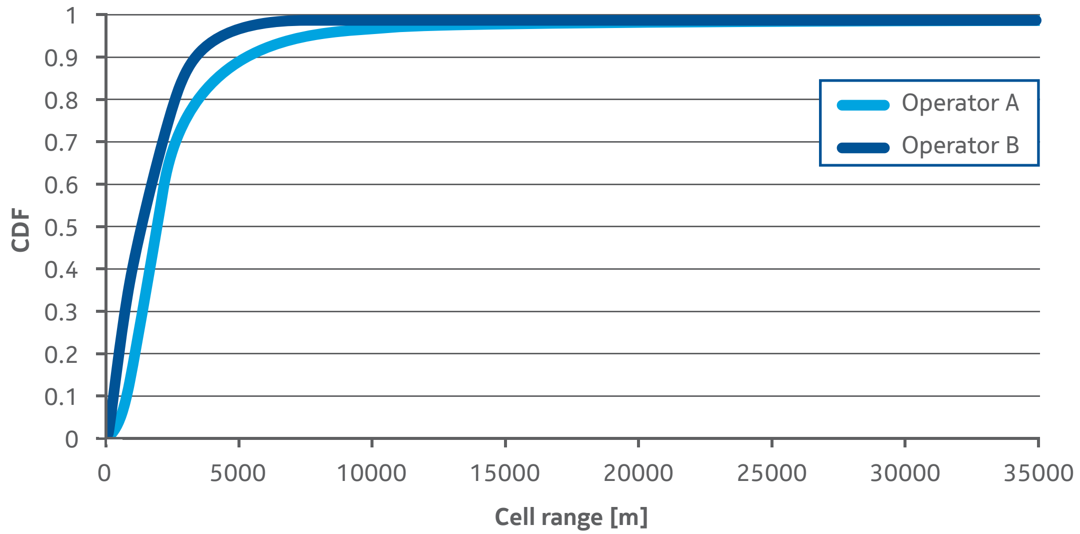


Figure 12 Cumulative Distribution Function as a function of cell range.



actually means to you. It is not only about the technology, but about a completely new way of doing business where data finally gets into the driver's seat. The holistic approach of Big Data enables you to uncover facts on entangled properties of system elements and the relations between them. By using the big data methods and tools, it is possible to infer the temporal characteristics of the telco market embedded into our reality composed of many contexts. Simply speaking, in response to the question of how the dynamics of the telco market trigger other businesses and how these other domains drive the telco market is a mission for the Big Data team in Nokia.

In this paper, the foundations of Big Data were presented, including methods and tools used in this field of knowledge, and supplemented with exemplary applications to telecommunications. This work positions the big data team in Nokia's strategy, which drives the temporal and future status of the telco market as a whole. By planning your professional path at the crossroads of BD and telecommunications, you can familiarize yourself with the steps necessary in becoming a specialist in Big Data, especially with our supervision.

We are now at an early stage of the BD era, and every Reader – directly or indirectly – participates in its flow. All of us are the source of data and only a select few have a chance to analyze them in their daily work, to shape the business of the future, and to create a better reality for our world.

References

[1] http://www.sciencedaily.com/releases/2013/05/130522085217.htm
[2] http://www.bbc.com/news/business-26383058
[3] https://sharenet-ims.inside.nokiasiemensnetworks.com/livelink/livelink/overview/D529590938.
[4] Cukier K., The Economist, Data, date everywhere: a special report on managing information, 2010, February, Retrieved from: http://www.economist.com/node/15557443
[5] http://db-engines.com/en/ranking
[6] http://www.forbes.com/sites/danwoods/2012/07/23/ten-properties-of-the-perfect-big-data-storage-architecture/
[7] http://nirvacana.com/thoughts/becoming-a-data-scientist/

About the authors

Graduate of Wrocław University of Technology in Faculty of Electronics, Teleinformatics. Experienced with software integration and verification, mostly in test automation area. Currently focused on LTE network features and data analytics aspects.

Sławomir Andrzejewski
Specialist, Network Engineering
MBB Customer Support

Post-graduate studies from Inje University, South Korea with undergraduate degree from the University of the Philippines in Electronics and Communications Engineering. Currently focused in area of Big Data Analytics, LTE load balancing and SON features support.

John Paul Torregoza
Specialist, Network Engineering
MBB Customer Support

Post-graduate studies from Wrocław University of Technology in Electronics. Interested in research & engineering work on physical-mathematical modeling and experimental techniques. Currently focused in area of Big Data Analytics for telco business optimization.

Ireneusz Jabłoński
Specialist for Data Mining and Machine Learning
MBB Customer Support

Graduate of Wrocław University of Technology in faculty of Electronics, information and communications technology. Experienced LTE engineer with deep radio features insight and system performance analysis. Recently, supporting the Big Data & Analytics streams.

Krzysztof Waściński
Radio Concept Owner
MBB Customer Support

Telecommunication System Engineering

LTE-Advanced – Mobile Broadband Network Technology of Tomorrow, Available Today

Grzegorz Olender
R&D Manager
MBB FDD LTE

1. Introduction

Long Term Evolution (LTE) mobile communication standard has been developed and specified by the 3GPP consortium in Release 8 (2008), with minor enhancements defined in Release 9 (2009). The LTE standard, even if marketed as 4G, does not fully satisfy requirements of International Telecommunication Union (ITU) for the fourth generation of mobile telecommunications technology. For this reason the 3GPP consortium has enhanced the LTE standard and published the LTE-Advanced (LTE-A) standard in Release 10 (2011), with additional enhancements in Release 11 (2012). The LTE-A is called a "True 4G" because it actually meets the ITU requirements for International Mobile Telecommunications-Advanced (IMT-Advanced) systems.

2. System requirements

IMT-Advanced (4G) is a term used by ITU for systems that offer new capabilities of IMT and go beyond IMT-2000 (3G). System requirements for IMT-Advanced were published by ITU in 2008. IMT-Advanced systems shall support low to high mobility applications and a wide range of data rates in accordance with the user and service demands in multiple user environments. IMT-Advanced shall also have capabilities for high quality multimedia applications within a wide range of services and platforms, providing a significant improvement in performance and the quality of service. [1]

Key features of IMT-Advanced [1] are:

- high degree of commonality of functionality worldwide while retaining the flexibility to support a wide range of services and applications in a cost-efficient manner
- compatibility of services within IMT and with fixed networks
- capability of interworking with other radio access systems
- high-quality mobile services
- user equipment suitable for worldwide use
- user-friendly applications, services and equipment
- worldwide roaming capability
- enhanced peak data rates to support advanced services and applications (100 Mbps for high mobility and 1 Gbps for low mobility were established as targets for research)

Key technical requirements of IMT-Advanced are [2]:

- support of scalable channel bandwidth up to and including 40 MHz (may be supported by single or multiple RF carriers), with a recommendation to consider extensions even up to 100 MHz
- peak spectral efficiency of 15 bps/Hz/cell in the downlink and 6.75 bps/Hz/cell in the uplink
- cell spectral efficiency of 3 bps/Hz/cell in the downlink and 2.25 bps/Hz/cell in the uplink
- control plane latency of less than 100 ms
- user plane latency of less than 10 ms
- handover interruption time of less than 60 ms

Peak spectral efficiency is defined as the highest theoretical data rate normalized by channel bandwidth under error-free conditions to a single mobile station when all available radio resources are utilized (that is, excluding radio resources used for physical layer synchronization, reference signals or pilots, guard bands and guard times). [2]

Table 1 IMT-Advanced requirements for peak spectral efficiency and resulting peak data rates. [2]

Direction	Peak spectral efficiency [bps/Hz/cell]	Peak data rate for 40 MHz channel bandwidth [Mbps]	Peak data rate for 100 MHz channel bandwidth [Mbps]
Downlink	15	600	1500
Uplink	6.75	270	675

Cell spectral efficiency is defined as the aggregate throughput of all users (the number of correctly received bits, that is, the number of bits contained in the Service Data Units (SDUs) delivered to Layer 3 over a certain period of time) divided by the channel bandwidth divided by the number of cells. [2]

Table 2 IMT-Advanced requirements for cell spectral efficiency. [2]

Test environment	Cell spectral efficiency [bps/Hz/cell]	
	Downlink	Uplink
Indoor	3	2.25
Microcellular	2.6	1.80
Base coverage urban	2.2	1.4
High speed	1.1	0.7

Channel bandwidth used for spectral efficiency calculations is defined as the effective bandwidth multiplied by the frequency reuse factor. **Effective bandwidth** is the operating bandwidth with the uplink/downlink ratio in case of Time Division Duplex (TDD) operation taken into account. **Frequency reuse factor** is the rate at which the same frequency is reused in the cellular network. It is denoted with 1/K where K is the cluster size, that is, the number of collocated cells associated with different frequencies. Practical K values are 1, 3, 4, 7, 9 and 12. In case of LTE, K is usually equal to 1.

The spectral efficiency is measured in **bps/Hz/cell**. The **data rate** can be calculated by multiplying the spectral efficiency by the channel bandwidth.

Control plane (C-Plane) latency is typically measured as the transition time from different connection modes, for example, from idle to active state. A transition time (excluding downlink paging delay and wire line network signaling delay) of less than 100 ms shall be achievable from idle state to an active state in such a way that the user plane is established. [2]

User plane (U-Plane) latency (also known as **transport delay**) is defined as the one-way transit time between an SDU packet being available at the IP layer in the user terminal/base station and the availability of this packet (Protocol Data Unit – PDU) at IP layer in the base station/user terminal. User plane packet delay includes delay introduced by associated protocols and control signaling assuming the user terminal is in the active state. IMT-A systems shall be able to achieve a user plane latency of less than 10 ms in unloaded conditions (that is, a single user with a single data stream) for small IP packets (for example, 0 byte payload + IP header) for both downlink and uplink. [2]

Handover interruption time is defined as the time duration within which a user terminal cannot exchange user plane packets with any base station. The handover interruption time includes the time required to execute any radio access network procedure, radio resource control signaling protocol, or other message exchanges between the user equipment and the radio access network. For the purpose of determining the handover interruption time, interactions with the core network (that is, network entities beyond the radio access network) are assumed to occur in zero time. It is also assumed that all the necessary attributes of the target channel (for example, downlink synchronization is achieved and uplink access procedures, if applicable, are successfully completed) are known at initiation of the handover from the serving channel to the target channel. [2]

Table 3 IMT-Advanced requirements for handover interruption times. [2]

Handover type		Interruption time [ms]
Intra-frequency		27.5
Inter-frequency	**within a spectrum band**	40
	between spectrum bands	60

IMT-Advanced defines the following **mobility classes** [2]:

• Stationary:	= 0 km/h
• Pedestrian:	> 0 km/h to 10 km/h
• Vehicular:	10 km/h to 120 km/h
• High speed vehicular:	120 km/h to 350 km/h

Table 4 IMT-Advanced test environments. [1]

Test environment	Mobility classes supported	Maximum speed [km/h]
Indoor	Stationary, Pedestrian	10
Microcellular	Stationary, Pedestrian, Vehicular (up to 30 km/h)	30
Base coverage urban	Stationary, Pedestrian, Vehicular	120
High speed	Vehicular, High Speed Vehicular	350

LTE-A supports mobility across the cellular network and is optimized for low mobile speed within the range from 0 to 15 km/h. Higher mobile speeds between 15 and 120 km/h are also supported with high performance. Mobility across the cellular network can be maintained at speeds from 120 km/h up to 350 km/h (or even up to 500 km/h depending on the frequency band). [3]

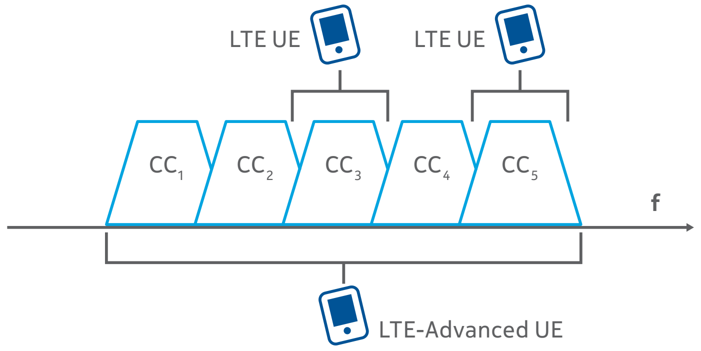
Table 5 Fulfillment of IMT-Advanced requirements in LTE (R8) and LTE-Advanced (R10) systems. [3], [4]

Requirement		IMT-Advanced	LTE (R8)	LTE-Advanced (R10)
Bandwidth		≥ 40 MHz	≤ 20 MHz ×	≤ 100 MHz ✓
Peak spectral efficiency	**Downlink**	15 bps/Hz	16 bps/Hz ✓	30 bps/Hz ✓
	Uplink	6.75 bps/Hz	4 bps/Hz ×	16 bps/Hz ✓
Latency	**Control Plane**	≤ 100 ms	50 ms ✓	50 ms ✓
	User Plane	≤ 10 ms	5 ms ✓	5 ms ✓

3. Carrier Aggregation (CA)

Maximum bandwidth of a single radio frequency (RF) carrier in LTE is 20 MHz. It was sufficient for the initial deployment of LTE, but it did not fulfill system requirements of IMT-A. In order to overcome this limitation and support higher data rates, the **Carrier Aggregation (CA)** [5] is introduced in the LTE-A. At the same time backward compatibility to earlier LTE releases is maintained, so that LTE User Equipment (UE) may access the network using a single RF carrier only, whereas CA capable UE in LTE-A may use more than one RF carrier at the same time to achieve higher data rates.

Figure 1 Carrier Aggregation (CA).

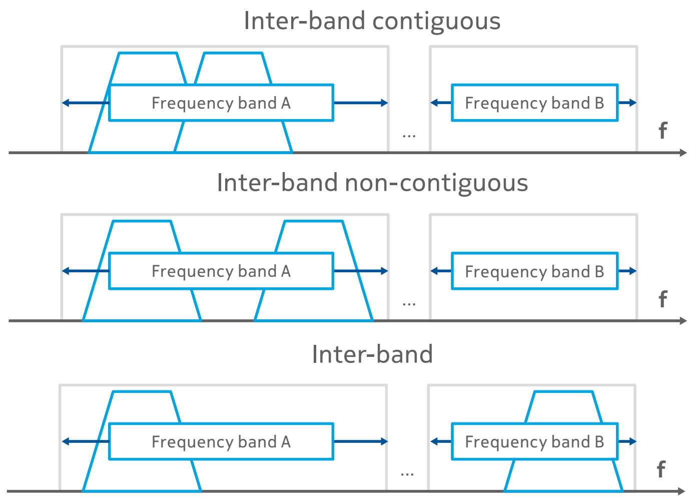


Each aggregated RF carrier is referred to as a **Component Carrier (CC)**. The CC can have a bandwidth of 1.4, 3, 5, 10, 15, or 20 MHz. A maximum of five CCs can be aggregated. Therefore, the possible total aggregated bandwidth is 100 MHz (5 CCs x 20 MHz).

In FDD operation CCs can have different bandwidths and the number of aggregated CCs can be different for DL and UL. In TDD operation the number of CCs and the bandwidth of CCs are the same for DL and UL.

Each CC includes Primary and Secondary Synchronization Signal (PSS and SSS), Physical Broadcast Channel (PBCH) and Reference Symbols (RS). That creates additional overhead with respect to Carrier Aggregation, however it is necessary to ensure backward compatibility to earlier LTE releases.

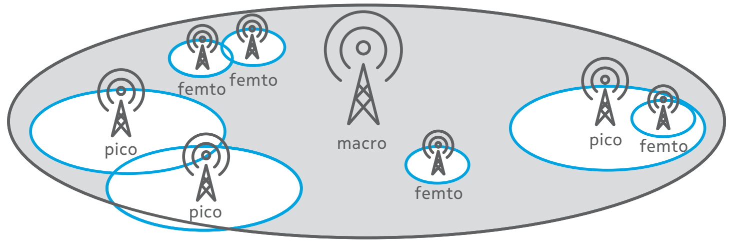
Figure 2 Carrier Aggregation (CA) deployment scenarios.



4. Enhanced Inter-Cell Interference Coordination (eICIC)

LTE employs the Load Indication procedure that can be used to exchange **Inter-Cell Interference Coordination (ICIC)** information between neighboring eNodeBs (eNBs). The purpose of ICIC is to prevent interference if neighboring cells operate with the same frequency. ICIC works with frequency resolution of Physical Resource Blocks (PRB) (that is, twelve consecutive sub-carriers each of 15 kHz). However, in networks where cells may have different sizes, the ICIC solution is not sufficient anymore, especially in case of overlapping cells that operate on the same frequency.

Figure 3 Example topology of heterogeneous cellular network.



For this reason LTE-A introduces **Enhanced Inter-Cell Interference Coordination (eICIC)** [6] with a concept of **Almost Blank Subframes (ABS)**. ABSs are dedicated subframes (1 ms long) scheduled without any data transmission and thus without corresponding control information. Their purpose is to prevent interference in case of overlapping cells that operate on the same frequency.

For picocells with the so-called Cell Range Extension (CRE), the ABSs are created within a macro cell. In this case the task of macro cell is to blank out certain subframes in which the underlying picocell schedules relevant information for the interfered UE, for example, signal measurements.

For femtocells with the so-called Closed Subscriber Groups (CSG), the ABSs are created within a femtocell. In this case the task of femtocell is to blank out certain subframes in which the overlying macro cell schedules relevant information for the interfered UE, for example, signal measurements.

The corresponding Load Indication procedure has been enhanced in the LTE-A with new eICIC-related parameters and is used for picocells. However, femtocells usually do not have X2 interface [7], so the only way to set up ABSs in femtocells is to configure them via O&M interface.

A challenge related to eICIC is a demand for tight time synchronization between the coordinated overlapping cells as the ABSs are defined at the subframe level.

5. Coordinated Multi-Point (CoMP)

Coordinated Multi-Point (CoMP) [8] operation is a range of different techniques that enable dynamic coordination of transmission and/or reception over geographically separated sites in order to enhance system performance and service quality. CoMP can be seen as a generalization of eICIC, with interference being converted into useful signal, especially at the cell edges where performance and quality may be degraded.

Joint transmission means that data is transmitted to a mobile station jointly from several points. Joint reception means that data is received from a mobile station jointly at several points. In general, both joint transmission and reception require low latency in the communication between network node and different antennas involved. Hence, in practice different sites may be connected together to form the so-called **centralized RAN**.

CoMP implies dynamic coordination among multiple, geographically separated transmission points where a transmission point is defined as a set of geographically collocated and correlated transmit antennas.

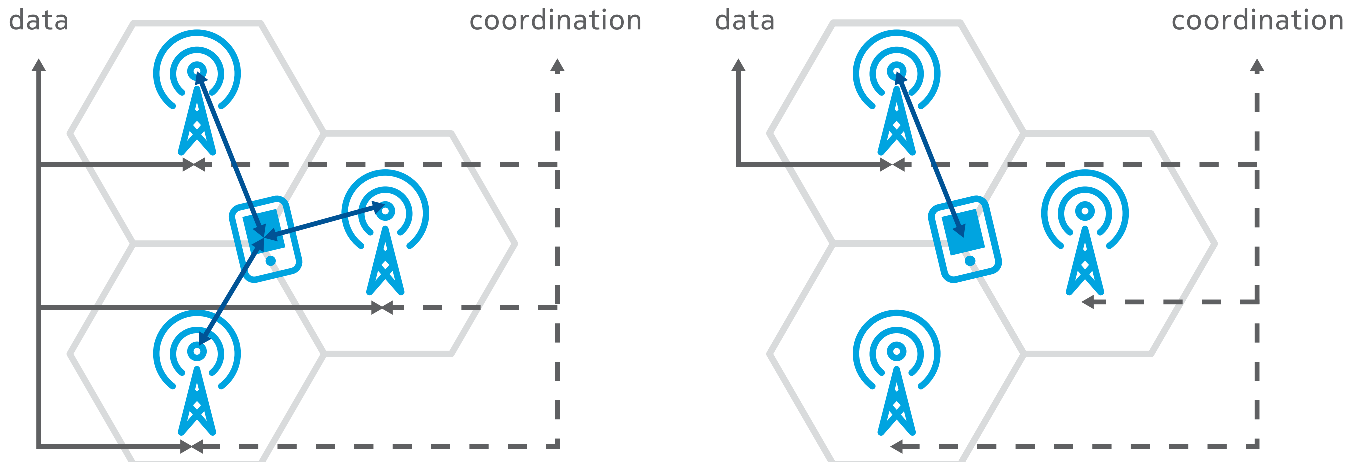
Specific techniques used for CoMP are different for the uplink and for the downlink. That results from the fact that eNBs are inter-connected within a network, whereas UEs are individual elements.

CoMP cooperating set is a set of points which are the subject of CoMP operation. The most important CoMP operation schemes are:

- **Joint Transmission (JP)/Joint Reception (JR)**: data is available at more than one point within the CoMP cooperating set and transmitted (DL) or received (UL) via multiple points simultaneously, for example, to improve signal quality and/or data throughput.
- **Dynamic Point Selection (DPS)**: data is available at more than one point within the CoMP cooperating set, however, data is transmitted (DL) via one point only, the transmission point may change from one subframe to another, for example, to select the best signal quality (for DL only).
- **Coordinated Scheduling (CS)/Coordinated Beamforming (CB)**: data is available at one point only within the CoMP cooperating set and transmitted (DL) or received (UL) via that point only. However, scheduling/beamforming decisions are made in coordination, within the corresponding CoMP cooperating set.

A challenge related to CoMP is a demand for highly efficient internal transport interface solutions in base stations that guarantee very high data rate (going beyond 10 Gbps) and low latency (below half of a millisecond) at the interface between baseband block and radio block.

Figure 4 Coordinated Multipoint (CoMP) schemes.



a) joint transmission
joint reception

b) coordinated scheduling
coordinated beamforming

6. Conclusion

Cellular networks are continuously evolving so new network technology requirements are emerging. One important example of this evolution is a generation of **heterogeneous networks (HetNet)** [9], where access points of different types and network cells of different sizes are utilized to offer flexible and complete wireless coverage. HetNet is a network with a complex interoperation between macro cells, small cells and, in some cases, WiFi network elements used together to provide sufficient coverage with handover capability between network elements.

Another important trend is utilization of **small cells** created by low power radio access nodes that operate in licensed and unlicensed spectrum with a range of:

- up to 10 meters (femtocell)
- up to 200 meters (picocell)
- up to 2 kilometers (microcell)

A typical macro cell may have a range of tens of kilometers.

Technology evolution related to heterogeneous networks or small cells is an important trend in the field of future mobile broadband. LTE-A with features like CA, eICIC and CoMP is one of the key system solutions that make it happen. LTE-A is designed for the mobile broadband network technology of tomorrow and is available today already.

References

[1] „Report ITU-R M.2135; Guidelines for evaluation of radio interface technologies for IMT-Advanced".
[2] „Report ITU-R M.2134; Requirements related to technical performance for IMT-Advanced radio interface(s)".
[3] „3GPP TR 36.912; Feasibility study for Further Advancements for E-UTRA (LTE-Advanced)".
[4] „3GPP TR 36.913; Requirements for further advancements for Evolved Universal Terrestrial Radio Access (E-UTRA) (LTE-Advanced)".
[5] „3GPP TR 36.808: Evolved Universal Terrestrial Radio Access (E-UTRA); Carrier Aggregation; Base Station (BS) radio transmission and reception".
[6] „3GPP TR 36.331: Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification".
[7] „3GPP TR 36.423: Evolved Universal Terrestrial Radio Access Network (E-UTRAN); X2 Application Protocol (X2AP)".
[8] „3GPP TR 36.819: Evolved Universal Terrestrial Radio Access (E-UTRA); Coordinated multi-point operation for LTE physical layer aspects".
[9] „3GPP TR 36.839: Evolved Universal Terrestrial Radio Access (E-UTRA); Mobility enhancements in heterogeneous networks".

About the author

I studied Communication Systems and Networks at the Wrocław University of Technology and the Technische Universität München. With over ten years of experience in system research and software development I have been engaged in automobile system software standardization, telematics systems integration, and radio modules development. Currently, I am working in the field of LTE software development. I am interested in software engineering, discrete mathematics, competence development, and cooperation with local universities. In my free time I practice triathlon.

Grzegorz Olender
R&D Manager
MBB FDD LTE

OBSAI and CPRI – Internal Transport Interfaces in Base Stations

Michał Koziar
Engineer, Hardware Integration
MBB Radio Frequency

Zdzisław Nowacki
Engineer, Hardware Integration
MBB Radio Frequency

1. Introduction

1.1. Cell types and their impact on BTS parameters

To describe the role of internal transport interfaces we have to start from a short introduction to the concept of cellular networks. Their key elements are Base Transceiver Stations (BTSs). The BTS architecture and parameters must be strictly adjusted to network operator requirements like cell area and capacity meant as the maximum number of voice calls and data transfer. **Figure 1** shows an example of a cellular network.

In this kind of network we can distinguish several types of cells:

- Pico cells (marked in blue) have a small range and are typically installed in dense urban areas with large number of voice calls and high data transfer.
- Micro cells (marked in navy blue) have a medium range and are dedicated to areas with medium voice and data traffic.
- Macro cells (marked in gray) provide ranges higher than micro cells.

The cell type has an impact on BTS requirements like transmitter power, number of antenna sectors, baseband block with DSP resources related to traffic processing e.g. channel coding. For example, a small-cell BTS mounted in a city center will have to handle a lot of voice calls and high data transfer. So, this type of BTS will contain a complex baseband with high DSP processing capability, but the RF transmitter may a have small output power. The opposite of a small cell is a macro cell. This kind of cells can be located in areas with low housing density which has an impact on the coverage area – the radius of such a cell can be 20 km or more. To cover a large area, a transmitter with high output power is needed. But in contrast to a BTS installed in a small cell, the baseband block can be less complex because of a lower traffic. The above-mentioned examples show that architecture of BTS must be scalable and modular. In 1G and 2G technology networks BTSs were based on an all-in-one architecture. Analog and digital electronics were installed in a large cabinet [8] placed in a special equipment room with air condition, backup battery etc. A relatively large area required for such a BTS installation would raise the cost of operation and limit installation location options. Thanks to miniaturization of electronic components, as well as boost in their efficiency, a distributed BTS architecture has been introduced for 3G networks.

Figure 1 Cellular network and cell types.

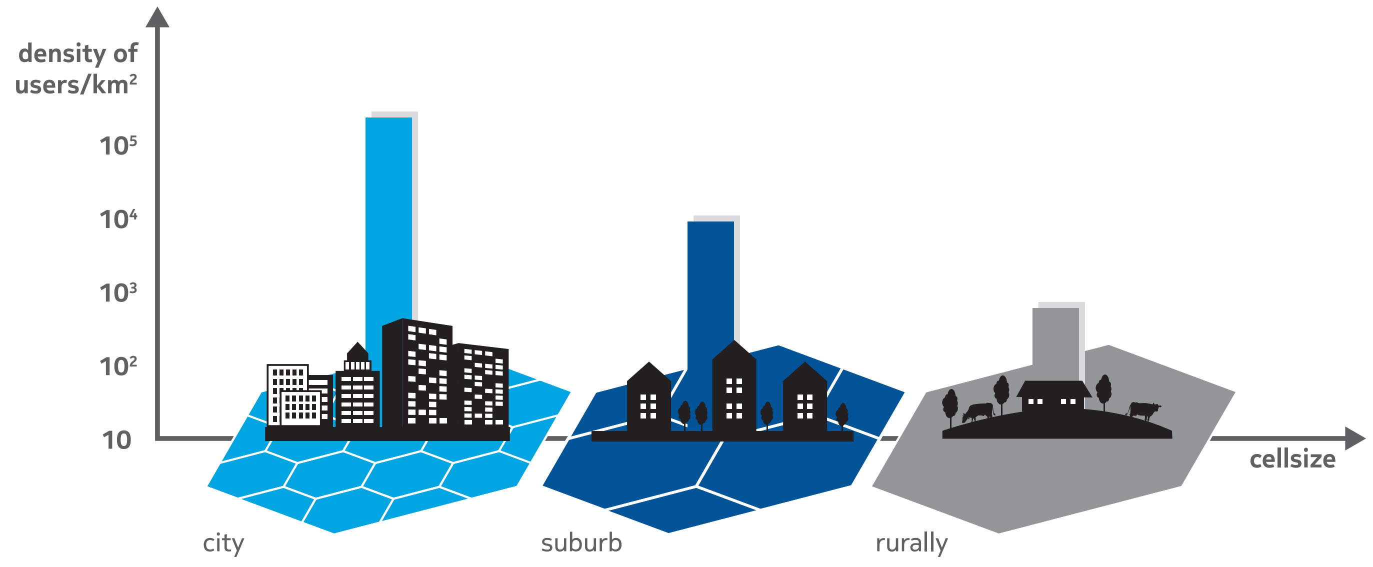
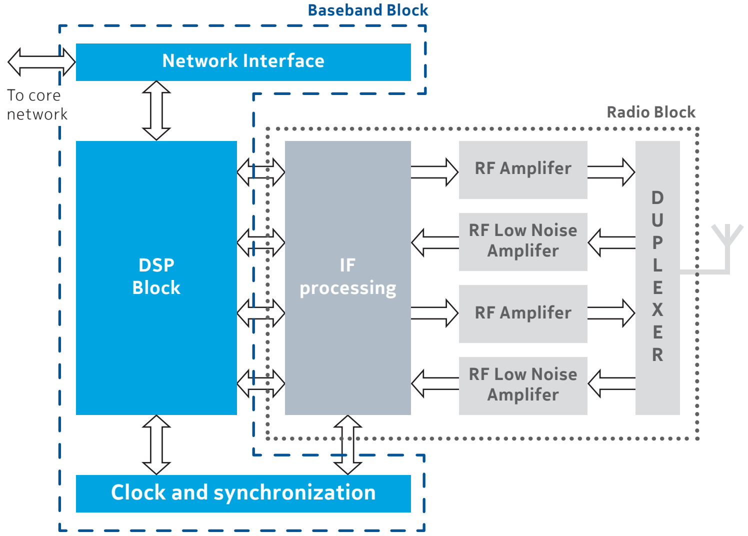


Figure 2 BTS block scheme.



1.1. Distributed BTS architecture

Figure 2 presents a block diagram of a BTS. The blocks are as follows:

- DSP Block – responsible for digital signal processing like scrambling or channel coding
- Clock and synchronization
- Network interface block – responsible for communication between the BTS and the rest of the network
- Intermediate Frequency block – responsible for downconverted RF signal processing like modulation and demodulation
- RF Amplifier in transmitter section
- RF Low Noise Amplifier in receiver section

These modules can be grouped into two main blocks:

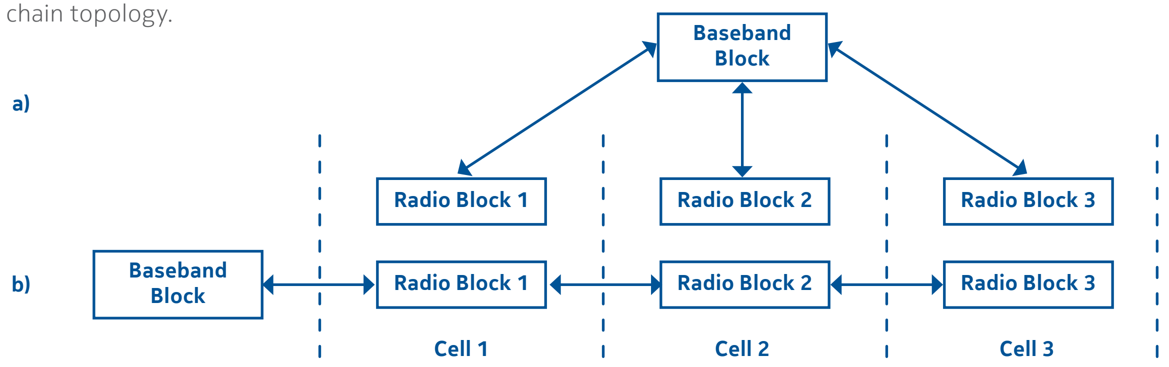
- Radio Block, which includes all functionalities related to RF and IF signal processing
- Baseband Block, which includes all functionalities related to DSP processing and connectivity with the rest of cellular network elements

Distributed BTS architecture is based on a physical separation of Baseband Block and Radio Block. Radio Block can be installed close to the antenna. Short cables from the RF amplifier to the antenna limit RF signal loss and increase the BTS power efficiency. There is required an efficient transport interface between these blocks.

Let us consider a specific case of network structure – cells located along a highway with low housing density. In this situation we can expect a rather low traffic and data transfer but a large radius of the cell. That means the BTS can be built with a Radio Block containing a high output power RF amplifier and a Baseband Block with low or medium DSP processing power. Two possible topologies are shown in Figure 3. In case of a chain topology, total capacity of air interface is limited by link capacity that connects Radio Blocks with the Baseband Block. Additionally, distance from the Baseband Block to the farthest Radio Block imposes certain restrictions. That will be discussed later in this article. For star topology, link capacity is not so critical. When traffic grows in only one cell, the network operator can add an additional Baseband Block. However, the star topology may require use of optical fibers with a bigger total length and more cable laying works. In practice, the number of configurations for interconnection between the Baseband Block and the Radio Block is much larger than presented in this article. More details can be found in sections 2.3 [2] and 3.3 [1].

As we can see, this kind of solution, based on modularity, offers scalability, easy network extension, and optimal infrastructure usage.

Figure 3 Topology of connection between Radio Block and Baseband Block for cells located along highway, a) star topology, b) chain topology.



2. Role of transport interfaces in BTS

The role of transport interface in the BTS is as follows:

- Data transmission for air interface
- Control and management of data transmission
- Synchronization of transport interface with air interface including compensation of delays in transport interface

2.1 Data transmission for air interface

Data for air interface has the biggest impact on the transport interface line rate. As an example, let us consider 10 MHz LTE signal. According to 3GPP TS 36.211 [3] the sampling rate for such a signal is 15.36 Ms/s. Assuming that one sample has the length equal to 16 bit and includes two symbols (I and Q), a result is that the data rate is 15360000 x 2 x 16 = 491.52 Mbps. For LTE 20 MHz, the sampling rate can be 30.72 Ms/s which makes the data rate equal to 938.04 Mbps. The number of used antenna carriers and the standard of radio interface result in typical data rates reaching the range of a few Gbps.

2.2. Control and management of data transmission

Remote access to BTS parameters (e.g. temperature of power amplifier in RF transceiver) and remote SW upgrade are important from the control and management point of view. The size of control data ranges over tens of Mbps. Typically, Ethernet data encapsulated into transport interface is used for control purposes.

2.3. Synchronization of transport and air interfaces

Synchronization of transport and air interfaces is very important for efficient operation of cellular network. The example from Figure 3b is to help describe the problem. Let us assume that the distance between each Radio Block and Baseband Block is 5 km. For such connections single mode optical fibers are used. Their propagation delay is around 5 μs per km. So, propagation delay for data from Baseband Block to the first Radio Block will be 25 μs and 75 μs for the last Radio Block.

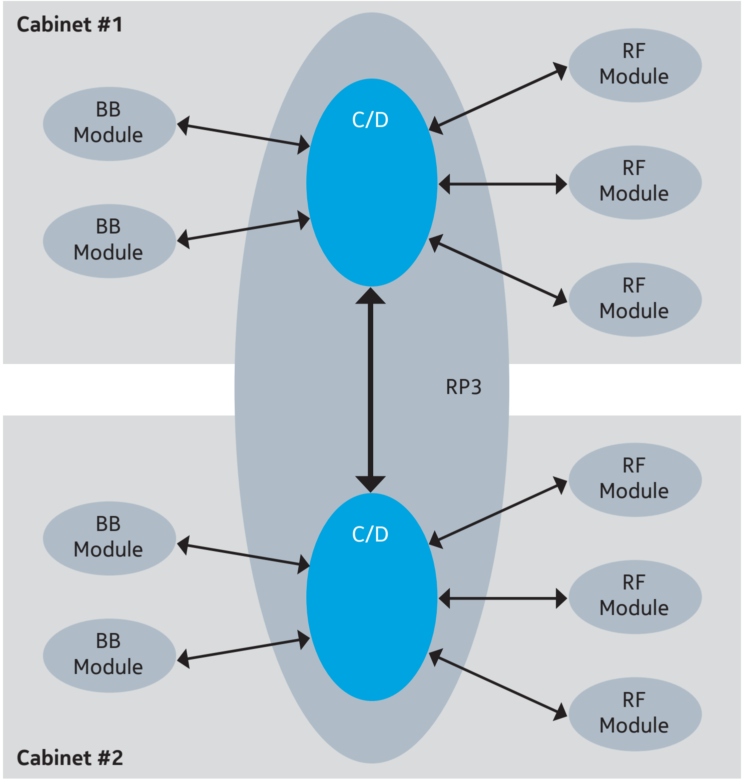
IQ data for radio interface is sent in the transport interface frame and is inserted into the air frame. All the cells are synchronized which means that air frame for cell 1 starts exactly at the same moment as in cell 2 or 3. Propagation delay causes IQ data in cell 3 to be 50 μs later than in cell 1. This delay needs to be compensated. Compensation is a complex process that starts with accurate measurements of propagation delays within the optical interface. Typical accuracy is in a range of single nanoseconds. After that each element of network sets its internal buffer so that it compensates latencies. For more detailed examples, see section paragraph 3.2.

3. Standards

3.1. OBSAI RP3-01

One of the interfaces used in Nokia BTSs is Open Base Station Architecture Initiative, Reference Point 3 (OBSAI RP3/RP3-01 [1]). This specification defines a number of solutions to problems that occur during transmission of digital data with synchronous electrical and optical links. The main points through which data is transmitted are called Nodes.

Figure 4 RP3 interface extended over two cabinets by connecting combiner and distributor modules (C/D) together.

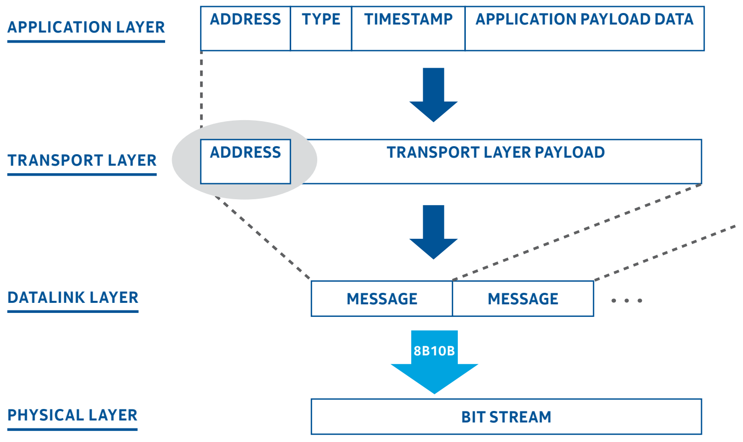


Reference Point 3 Specification defines three basic configurations:
a) point to point – modules are connected with the use of one or more RP3/RP3-01 links
b) chain – modules are connected in chain with the use of one link between the modules
c) point to multipoint – star configuration

Various combinations of topologies mentioned above might be used. In addition, the specification allows the use of a combiner and distribution modules.

The basic unit of information sent by links RP3/RP3-01 is the Message. It is composed of a header (address 13 bits, type 5 bits, time stamp 6 bits) and payload (16 bytes).

Figure 5 Layers in OBSAI RP3/RP3-01.



Address defines the Node where the Message must arrive. If the Node mediates communication between the participating nodes, the communication is based on the routing map and the Message is passed on.

Messages are divided into two groups:
a) IQ data messages (Antenna Carrier data)
b) Control messages (used for configuring the Node and transmitting other protocols e.g. Ethernet)

20 IQ messages and one control message form a Message Group (MG). After that the K.28.5 mark is sent for synchronization issues. After 1920 Message Groups (in case of line rate 1x – for details, see Table 1) are formed and sent, the frame continues with the K.28.7 mark – the end of Frame. Frame duration is 10 ms.

We can see that 20% of the data sent is lost in favor of service link (timing and control data).

Figure 6 RP3/RP3-01 Frame (Line rate 1x).

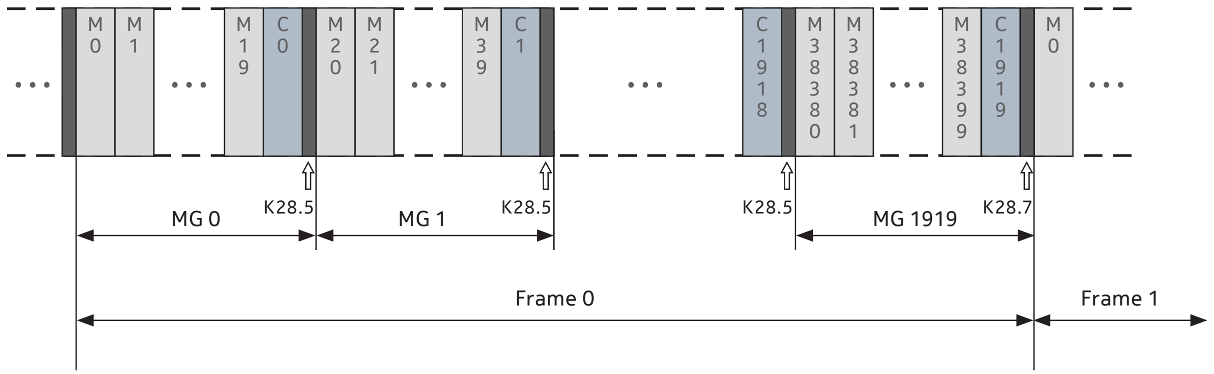
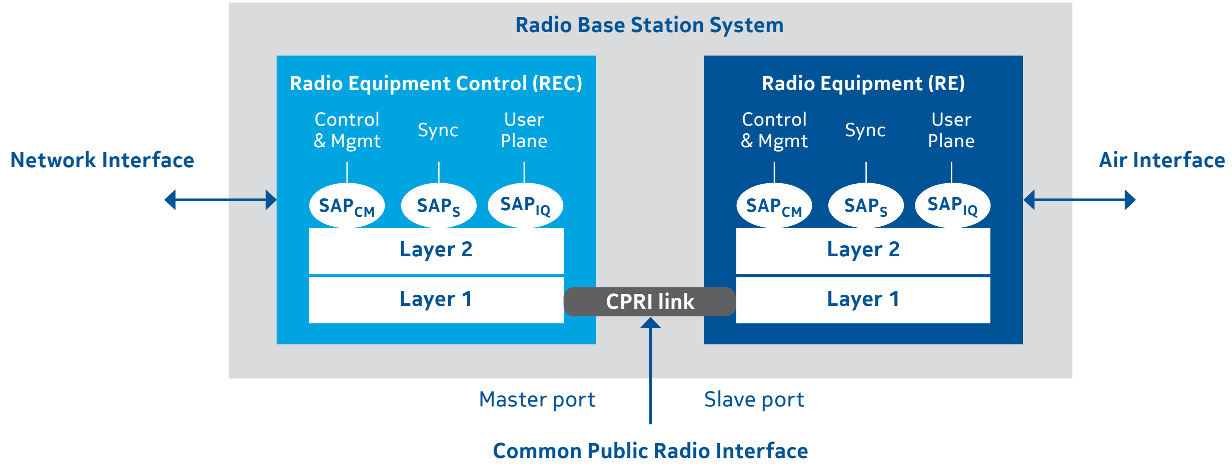


Table 1 Relation of link rate and data capacity.

Link rate	Link speed	No of Msgs Groups in Frame	No of IQ data msgs in Frame	IQ data transmission	Link capacity without 8b/10b coding	Total Link capacity
	Mbit/s		bytes/frame	bit/s	%	%
x1	768	1920	38400	491520000	80	64
x2	1536	3840	76800	983040000	80	64
x4	3072	7680	153600	1966080000	80	64
x8	6144	15360	307200	3932160000	80	64

Figure 7 Architecture of BTS according to CPRI.



3.2. CPRI

The other interface standard dedicated to BTS is Common Public Radio Interface (CPRI). The standard defines division of a BTS into two blocks shown in Figure 7:

- Radio Equipment Control (REC) that plays the role of Baseband Block
- Radio Equipment (RE) that plays the role of Radio Block

The structure of data in CPRI is shown in Figure 8. CPRI frame duration is 10 ms, the same as in OBSAI. CPRI frame consists of 150 hyperframes with index Z = 0...149. Each hyperframe consists of 256 basic frames with index X = 0...255. Each basic frame consists of 16 words with index W = 0...15. The length of a word depends on the CPRI line rate. For 614.4 Mbps, the word length is 8 bits but for line rate 9830.4 Mbps the word length is 128 bits. More details can be found in section 4.2.7 [2].

In contrast to OBSAI, CPRI does not send information about the type of data or about the final destination of data. CPRI strictly defines location of each type of data. For example, Ethernet payload encapsulation is defined by pointer p which can be flexibly configured by setting control byte Z.194.0. In general, each type of additional headers containing information about type and address causes CPRI for the same line rate as OBSAI to have higher capacity for IQ and control data. Currently, CPRI's maximum line rate is 12165.12 Mbps, compared to 6144 Mbps in OBSAI. More information about differences between CPRI and OBSAI can be found in [4].

3.2.1. Propagation delays

Each optical link has its own propagation delays: downlink T12 and uplink T34 that are strictly related to transport interface. Additionally, processing delays T2a and Ta3 need to be included into calculations. They are the sum of latencies in analog part of RE (i.e. filters, amplifiers) and latencies in digital part.

Figure 8 CPRI frame hierarchy.

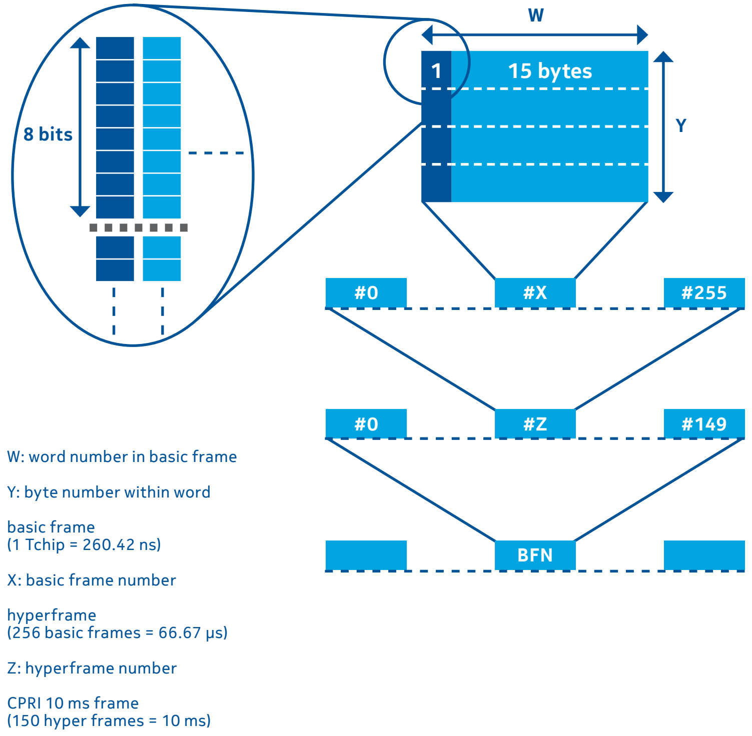


Figure 9 Definition of points for delay measurement.

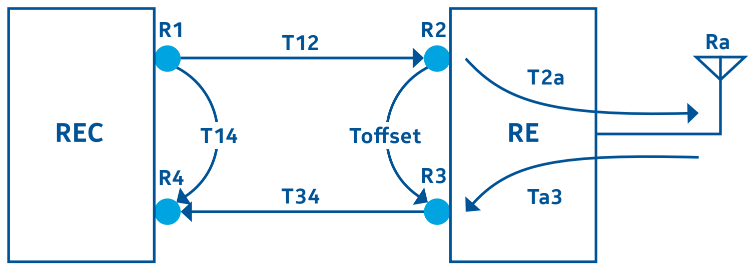
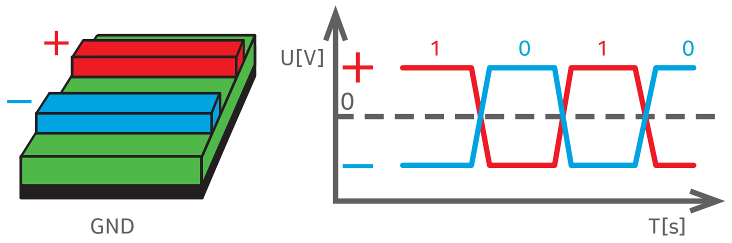


Figure 10 Differential transmission in time.



4. Physical layer

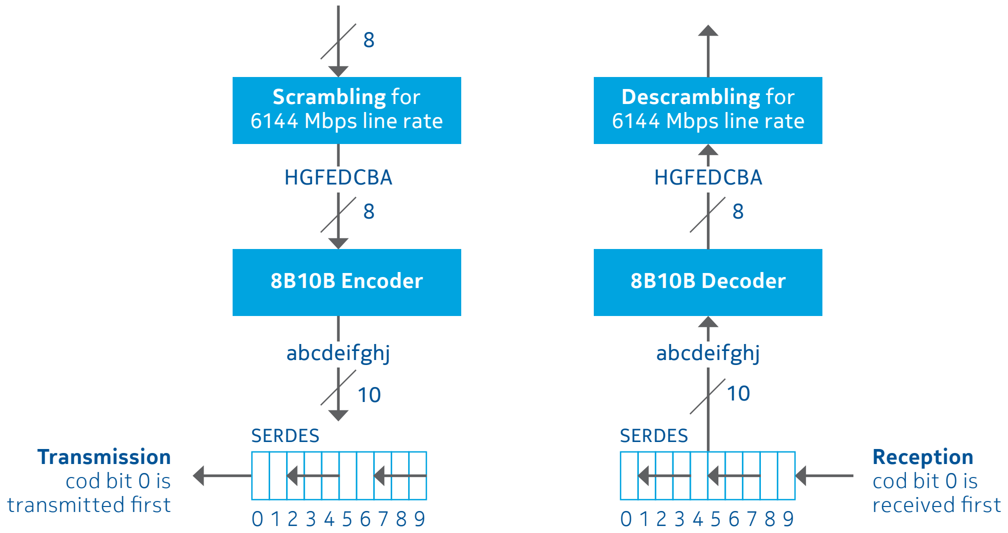
To send any data from point A to B a communication medium is needed. This medium forms the physical layer. The communication interfaces OBSAI and CPRI, described above, use the physical layer with a similar construction. The basic element is a link. It may be electrical and optical.

4.1. Electrical link

With an electrical link, connection provides us with two pairs of wires: one to receive, the other to transmit information. Let us call one of the wires "+" and the other one "-". The voltage difference between the positive and negative is relative to 0 (GND). If it is positive, we call this state a logical 1; if it is negative, we call this a logical state 0. It is a differential transmission – we study a potential difference. More information about differential signaling can be found in [7].

With the information thus obtained (0 or 1) our signal goes to a circuit called the Serializer/Deserializer (SerDes). Because the data is transmitted in a serial way – bit by bit, we have to group it somehow. The circuit SerDes puts our individual data bits together into bytes. A benefit is that processors involved in the processing of this information do not require high speeds and consequently consume less energy. Our first readable unit of information contains 10 bits. This technology uses 8b/10b coding, which enables serial data transmission through one pair of wires with high speed and without the use of additional clock line. So the data that comes out of the SerDes gets on the circuit, which aims to decode the 10 bits to 8 bits of information. This information is needed for building a frame structure of RP3 or CPRI standard.

Figure 11 SERDES, 8B10B encoder/decoder, scrambling in high speed receiver and transmitter.



One may be wondering how to find out which bit is the first one. This is the role of the 8b/10b decoding circuit. Each subsequent byte is stored in a specific way. To describe the first bit, the circuit is calculating checksum of bits. More details about this can be found in the 8b/10b encoder/decoder specification [5].

For OBSAI interface with the speed above 6 Gbps, scrambling is used. Scrambling is the process of multiplying each of the transferred bytes (does not apply to K-marks) by the constant bit pattern. That serves to minimize crosstalk between electrical links.

4.2. Optical link

Data transmitted by electrical link may be converted into light. That task is entrusted to Small Form-factor pluggable (SFP) [6]. SFP is not involved in any way in the encoding or decoding of data. It can only indicate a lack of data on an optical link. A standard optical module uses two fiber-optic cables per one link. There are also bidirectional SFPs, which send data in both directions using a single optical fiber. For such a transmission two different wavelengths are used, one for each signal direction. A fiber optic cable may have a length of up to 40 km.

5. Conclusions

Interfaces described in this article are used to transfer data between BTS modules. Due to the constant growth in demand for data transmission, i.e. future 5G technology or wireless network evolution towards Cloud – RAN architecture, the transport interfaces will need to be more and more efficient. In Cloud – RAN architecture the role of Baseband Block will be implemented in centralized computing data centers connected with a large number of Radio Blocks by Optical Transport Networks (OTN) that utilize CWDM and DWDM technology.

Commercially available FPGA chips include SerDes that works with the line rate up to 30 Gbps. Rising line rates force semiconductor manufacturers to increase research and development efforts in order to find new technologies that will allow us to exceed physical limitations of transmitting through a copper medium.

References

[1] OBSAI RP3-01
[2] CPRI Interface Specification V6.1 http://www.cpri.info/downloads/CPRI_v_6_1_2014-07-01.pdf
[3] 3GPP TS 36.211
[4] Chrisian Lanzani "Open Base Station Architecture: Can Standardization can enable true innovation?" http://www.mti-mobile.com/wp-content/uploads/2012/10/OBSAI_CPRI_Tutorial_and_Primer_ver02.pdf
[5] Lattice Semiconductor 8b/10b Encoder/Decoder http://www.latticesemi.com/~/media/LatticeSemi/Documents/ReferenceDesigns/1D/8b10bEncoderDecoder-Documentation.pdf?-document_id=5653
[6] AFBR-57J5APZ Datasheet http://www.avagotech.com/docs/AV02-0671EN
[7] Differential Pair Transmission Lines http://www.westmichigan-emc.org/archive/2014%20IEEE%20Bill%20Spence%20Diff%20Pairs.pdf
[8] Example BTS based on all in one architecture https://en.wikipedia.org/wiki/Base_station_subsystem#/media/File:Deutsches_Museum_-_The_guts_of_a_GSM_cell_site.jpg

About the authors

Both authors work as engineers in Hardware Integration and Verification team. In their daily work they deal with OBSAI and CPRI interface integration. Our work is related to BTS prototypes and focuses on aspects of the testing of hardware and software associated with transport interfaces, including reliability and performance.

Michał Koziar
Engineer, Hardware Integration
MBB Radio Frequency

Zdzisław Nowacki
Engineer, Hardware Integration
MBB Radio Frequency

LTE Global Verification – Testing In End-to-end Environment

Radosław Idasiak
R&D Manager
LTE Feature Verification

1. Introduction

In this article we describe how Nokia implemented feature development process and how our engineers examine the Long Term Evolution (LTE) system in an End-to-End (E2E) environment in order to achieve proper software quality and deliver clear information about new functionalities of the product to the customers.

First of all – everything is a feature. The Institute of Electrical and Electronics Engineers defines the term feature in IEEE 829 Standard for Software and System Test Documentation as:

“A distinguishing characteristic of a software item (e.g., performance, portability, or functionality).”

In our case it means that all of us work on certain functionalities, named features which represent a specific function in our LTE product. Features are functionalities desired by our customers based on their requirements or market demands.

What is important to understand is the fact that all the teams involved in product development and software life cycle development work on the same features. We all have the same goal – creating new functionalities in the system, by designing specifications, developing the code, testing the entity and the system. We all use the same documentation that delivers feature description, user scenarios, references to other parts of the system and requirements for the hardware. The entire process is presented in **Figure 1**.

System Architecture designs feature specification, usually based on external documents (i.e. 3rd Generation Partnership Project – 3GPP) or customer’s requirements. Their goal is to describe in detail the purpose of the functionality, all scenarios, interaction with other elements and usage possibilities by taking into account the already existing product interfaces and components.

Feature Teams are responsible for feature development based on specification provided by System Architects. They test functionalities which can work with real or simulated network elements on an entity level.

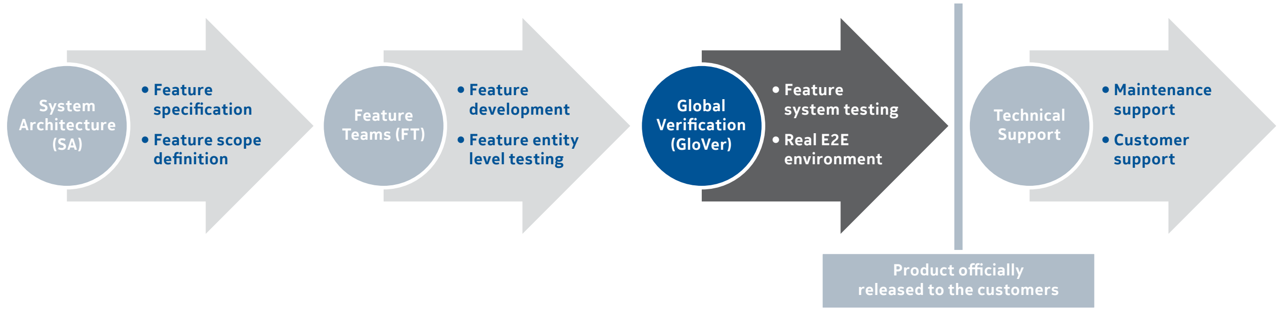
Global Verification performs feature testing in the end-to-end environment with real elements, starting form User Equipment (UE) and ending on specific Core Network (CN) elements. The goal is to test and verify the whole system behavior in a given scenario. Here we have system test execution. Global Verification organization is mainly responsible for testing new feature, but it also supports the maintenance phase together with Technical Support teams.

All feature requirements are taken into consideration. The test environment reflects the customer’s live network conditions and end user scenarios. This kind of testing can usually take place in the laboratories. Sometimes there’s a need to run specific cases in the field, where all Base Stations are deployed as in normal operator’s network. Responsibility given to this unit is very big. Global Verification is the last organization that tests the product before the official, commercial release into the market. We need to ensure that the coverage and tested scenarios deliver reliable results in order to make the final decision about the product release. Working with the LTE product version that will be available i.e. one year from now is exiting, but also brings certain challenges in normal day-to-day activities.

In order to better reflect the customer’s configurations and requirements, our engineers keep in continuous contact with customer teams around the world. This allows us to always get the most accurate information about the required hardware or configuration values. As a result we’re able to prepare the whole test environment with almost identical conditions as our customer’s real network is accustomed to. Based on this and feature specifications, testers create test procedures that are executed to check system behavior. Each tester is recognized outside of the Global Verification organization as the main representative for system testing of assigned functionality.

Technical Support works on all tickets reported by our customers after our product is available in their network. Technical Support is responsible for all levels of on-site and remote support, so in case something is not working properly, all customer’s requests and escalations are reported to this organization.

Figure 1 Feature development process flow.



2. Projects

Nokia runs several projects within the MBB LTE organization. In this article we will focus on following ones:

- RF-Sharing
- Trace Management

To cover the whole project scope from the system's perspective we need to ensure that all requirements are covered in our laboratory. Each engineer works on dedicated test equipment that covers a wide range of different elements and tools. By default all testers are equipped with:

- Set of UEs (usually newest smartphone models, or chipset vendor's prototypes) that allow testing newest functionalities implemented on network elements.
- LTE Evolved Node B (eNB) in different hardware (HW) variants and configurations to cover wider customer setups.
- Servers for logging data.
- Programmable attenuators and shieldboxes to reflect live network behavior.

The test area assigned HW may vary, but each engineer can work independently.

2.1. RF-Sharing

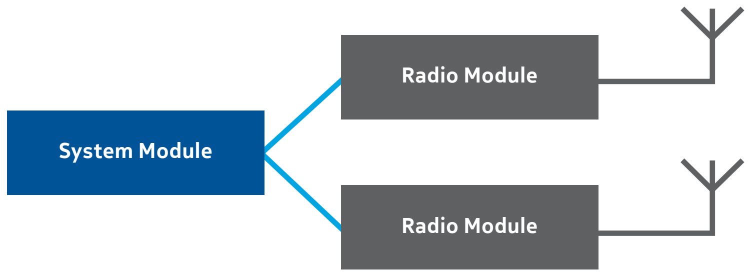
The RF-Sharing project aims to test features that allow Global System for Mobile Communications (GSM), Wideband Code Division Multiple Access (WCDMA) (used as a synonym for Universal Mobile Telecommunications System (UMTS)) and LTE to work on the same radio equipment. This introduces OPEX savings for customers and offers seamless LTE migration from currently existing and deployed Second-Generation Wireless Telephone Technology (2G) and Third Generation of Mobile Telecommunications Technology (3G) networks. We just need to add one small part of the HW to the already available 2G Base Transceiver Station (BTS) or 3G Node B (NB) to have LTE network.

Base stations are made up of System Modules and Radio Modules. This is one of the most basic HW configurations but it can also include:

- Extension modules inside of System Module, so in the end we can have bigger Base Station capacity or better performance.
- Antenna Line Devices (i.e. Remote Electrical Tilt (RET), Mast Head Amplifier (MHA)).

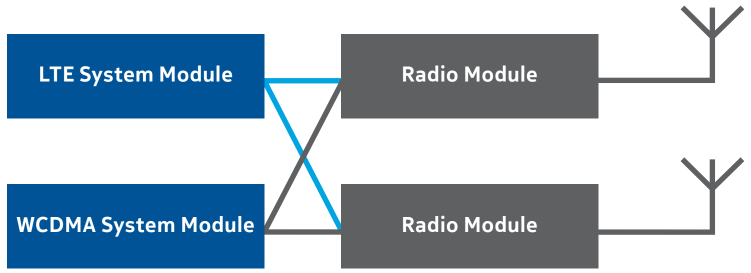
The System Module provides baseband processing, control and transmission functionality. The Radio Module is a stand-alone, technology independent, fully operational transceiver unit with a wideband transmission and reception technology inside that uses multi-carrier power amplifiers and wideband receivers. A basic overview is shown in Figure 2.

Figure 2 Example of Base Station HW configuration.



In order to introduce RF-Sharing and deploy LTE network on already used frequency bands, we can add a new System Module. An overview of the configuration is presented in Figure 3.

Figure 3 Basic RF-Sharing HW configuration (3G-4G example).



Remember – we can have more Radio Modules in such configurations which allow more cells to be defined or to cover more band combinations. Everything depends on our customer's requirements. Different Radio Access Technologies can operate simultaneously with carrier signals being transmitted and received via the same radio frequency unit. Multiple signals are transmitted through a broadband power amplifier, which shares its power source with defined carriers.

From the operator's point of view such configurations are seen as two separate logical network elements with common radio resources. Each radio technology has dedicated logical backhaul connections to radio controllers or other Core Network elements. Radio Resources are shared between both technologies in the same frequency band (both Radio Access Technologies (RAT))and use a common antenna system connected to the shared radio resources). The requirement that both system modules are synchronized together to share common radio resources is very important.

Figure 4 20 MHz carrier configuration for 3G-4G RF-Sharing – example 1.

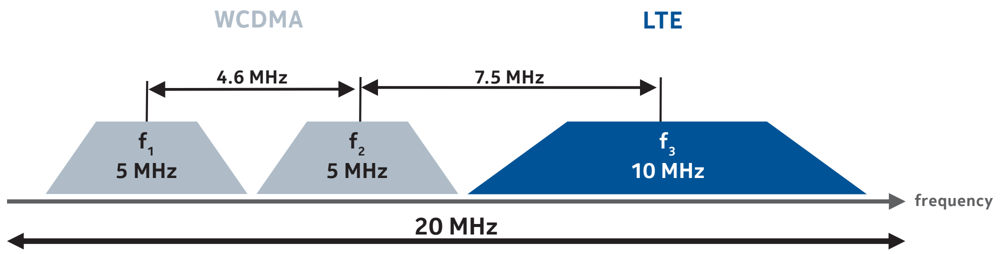
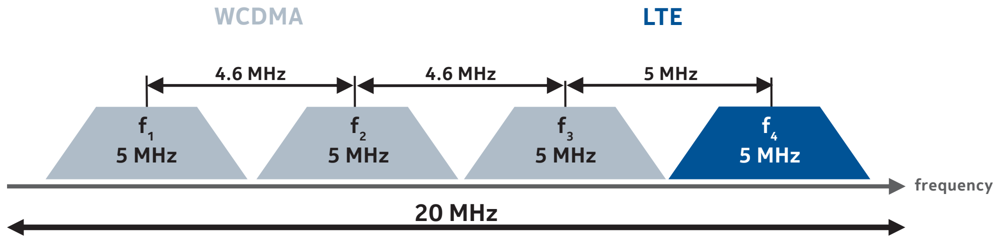


Figure 5 20 MHz carrier configuration for 3G-4G RF-Sharing – example 2.



In order to test it thoroughly, we need to follow the customer's configurations and requirements. To realize this, we can monitor the customer's network and obtain necessary information about deployed configurations. But the HW is not the only thing that we must focus on here. In order to reflect real life usage different kinds of parameters must be taken into account. Therefore our scope covers both areas. In the first phase of testing we focus on HW variants and all possible configurations. They are setup in the laboratory and tested to see if all connected units are properly detected. Here we need to follow the cabling description and OBSAI RP3, optical or Ethernet interfaces specification.

The next step covers the most basic part – software update. The aim of this test is to check whether all Base Station components faultlessly function with the newest available software (SW). This is checked for both cellular technologies. The right combination of SW is crucial here. Customer delivery is always released with SW packages which cannot be changed. System testing needs to check if internal communication between System Module and Radio Module functions properly. In the end, the entire configuration connects to other network elements and service is successfully provided.

Base station and other network elements have different kinds of logs for all internal applications and processes. Following 3GPP specifications together with internal feature descriptions we are always able to check if everything is working correctly. ISO/OSI model, LTE, WCDMA, GSM interface specification must be well understood. Using internal Base Station sniffing mechanisms or external monitoring services, we analyze the message flow and compare it with the specifications. All layers are checked starting from the physical side and ending on the application layer.

During configuration management we test different parameter value variants, cell setups, operational bandwidths, antenna power output and many others. The operational bandwidth that limits the number of carriers which can be handled simultaneously by single radio module pipe must be thoroughly investigated. It is very important to remember that all carriers must stay within the bandwidth designed for the Radio Module as each customer has a dedicated frequency range that can be used to deploy the service. This determines how many cells and which bandwidth values can be used. A simple example is shown in Figure 4 and Figure 5. Note that the carrier configurations depend on the HW variant and on the permitted bandwidth of the Radio Module.

Each Radio Module is equipped with a single Multi Carrier Power Amplifier (MCPA) per antenna pipe. While testing our customer's configurations we also check the cell power output values to see if both technologies function without conflicts. As in case of operational bandwidth, here the maximum output power at the antenna connector also depends on the HW variant. The maximum output power of the MCPA is shared among defined carriers, therefore it must be aligned with the number of carriers handled by the given MCPA. So if certain Radio Module supports 60W output power, we can divide it between technologies under the condition that the sum is equal to this value. Testing a radio signal on an antenna line with dedicated measuring HW on the physical layer and performing different call processing scenarios provides the final test results and confirms if everything functions properly.

As mentioned earlier, the Operating Support System (OSS) sees each configuration as a separate network element and therefore it requires dedicated tests for both technologies to verify if Operations & Maintenance (O&M) and Call Processing (CP) work correctly. Independent configurations based on providing parameters checks whether all values are correct. In order to thoroughly test the independence of both configurations we need to perform different site resets, modules (un)block or cells (un)lock scenarios starting with the LTE Base Station, where on the other (2G BTS or 3G BTS) we have some services running in parallel i.e. data calls. In the end we can test it in reverse in order to ensure ourselves that there are no consequences. The same approach is used for Fault Management, Performance Management or Call Processing scenarios. To test the interaction between both technologies all scenarios that we consider to be important for customers are in our test scope i.e. Voice over LTE (VoLTE) calls followed by performing Single Radio-Voice Call Continuity (SRVCC), external Network Assisted Cell Change (eNACC) and different types of handovers. Verification always includes counters values check in our dedicated OSS element, the same way as customer monitors its own network.

All the above tasks require specialist knowledge of cellular networks including 2G BTS, NodeB, eNodeB, Radio Network Controller (RNC), Serving GPRS Support Node (SGSN), General Packet Radio Service (GPRS) Gateway GPRS Support Node (GGSN), Base Station Controller (BSC), Mobility Management Entity (MME), Serving Gateway (S-GW), Packet Data Network Gateway (P-GW) and many others.

RF-sharing combines multiple radio technologies together and guarantees more efficient hardware utilization through shared resources. Therefore, an RF-sharing engineer faces the challenges that come from multiple aspects related to mobile communications. As a consequence, working with the RF-sharing solutions reveals an opportunity to spread the knowledge, practice and experience among all key elements of LTE-A, WCDMA and GSM as well.

Michal Kucharzak
Engineer, LTE Feature Verification

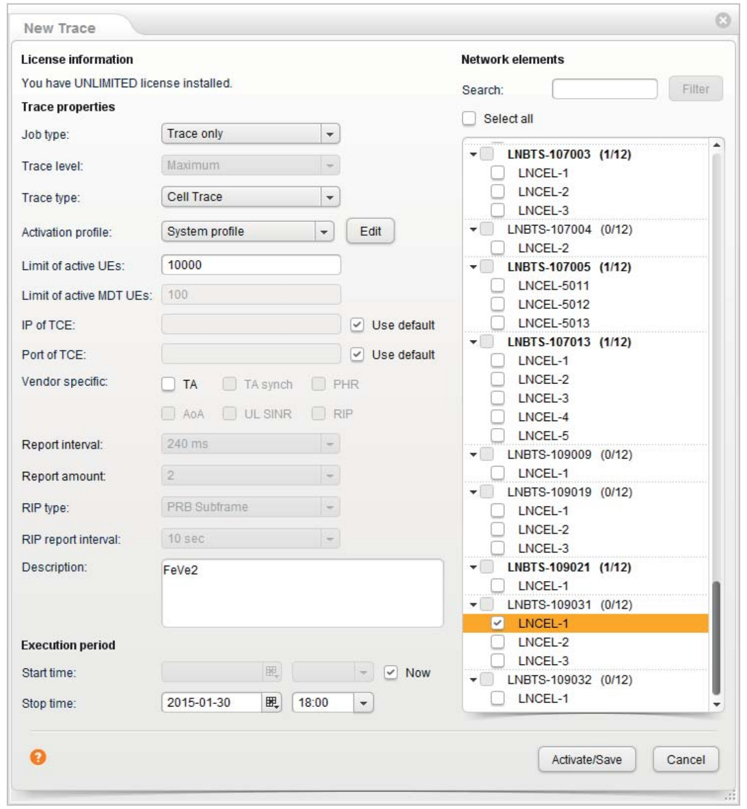
2.2. Trace Management

Features grouped under Trace Management functionalities provide a unique perspective when monitoring networks. This allows the operator to see exactly what happens in the network on different communication levels. The customer has the possibility to start the cell or subscriber trace and explore whole network behavior. Testers try to reflect subscriber behavior from the normal live network. Afterwards they check whether all messages were properly transferred between UE, eNB Evolved Packet Core (EPC) and other elements according to the usage of the OSS solution.

Trace testing involves a short procedure before preparing the whole network configuration. First we need to add certain parameters directly to the eNB, in order to activate the feature. Then we need to configure the Trace Collection Entity application on our OSS system. Activation involves selection of trace type:

- LTE Cell Trace supports tracing of the Layer 3 contents of S1-AP, X2-AP and eUu RRC messages of Control Plane. Additionally via the vendor specific extension we also test parts of the MAC and RLC information. Note that the user plane is not traced. Before collecting the network messages cell selection must be performed.

Figure 6 Example of Cell Trace configuration on Trace Collection Entity.



License information
You have UNLIMITED license installed.

Trace properties
Job type: Trace only
Trace level: Maximum
Trace type: Cell Trace
Activation profile: System profile
Limit of active UEs: 10000
Limit of active MDT UEs: 100
IP of TCE: ☑ Use default
Port of TCE: ☑ Use default
Vendor specific: □ TA □ TA synch □ PHR
□ AoA □ UL SINR □ RIP
Report interval: 240 ms
Report amount: 2
RIP type: PRB Subframe
RIP report interval: 10 sec
Description: FeVe2

Execution period
Start time: ☑ Now
Stop time: 2015-01-30 18:00

Network elements
Search:
□ Select all
□ LNBTS-107003 (1/12)
□ LNCEL-1
□ LNCEL-2
□ LNCEL-3
□ LNBTS-107004 (0/12)
□ LNCEL-2
□ LNBTS-107005 (1/12)
□ LNCEL-5011
□ LNCEL-5012
□ LNCEL-5013
□ LNBTS-107013 (1/12)
□ LNCEL-1
□ LNCEL-2
□ LNCEL-3
□ LNCEL-4
□ LNCEL-5
□ LNBTS-109009 (0/12)
□ LNCEL-1
□ LNBTS-109019 (0/12)
□ LNCEL-1
□ LNCEL-2
□ LNCEL-3
□ LNBTS-109021 (1/12)
□ LNCEL-1
□ LNBTS-109031 (0/12)
☑ LNCEL-1
□ LNCEL-2
□ LNCEL-3
□ LNBTS-109032 (0/12)
□ LNCEL-1

- LTE Subscriber Trace provides detailed subscriber oriented information for one or more specific mobile(s) at call level. Testing for this functionality is even done at equipment level. It can be done for a limited period of time for specific analysis purposes, e.g. for root cause determination of a malfunctioning mobile phone, advanced troubleshooting, resource usage and quality optimization, radio frequency coverage control and capacity improvement, dropped call analysis or E2E procedure validation. Such tracing types use International Mobile Subscriber Identity (IMSI) or International Mobile Equipment Identity (IMEI) data to collect all required information. For this reason we need to also communicate with EPC elements in order to obtain IMSI or IMEI. The IMSI information needs to be provided from the core network element based on 3GPP defined messages and is included in the trace data content. This is why we need to check whether the eNB is sending unique trace reference to the MME via the S1 connection during the tests. After this we check if upon receiving such message the MME resolves the IMSI and the IMEI of the given call and sends the IMSI, IMEI numbers together with the trace reference to the Trace Collection Entity. Finally, we check if Trace Collection Entity combines the information from eNB and MME correctly and if all traced data are presented in the right way.

Trace verification is done according to internal specification and the following 3GPP documents [1, 2, 3]. The final trace results are sorted by received messages per connection (e.g. using X2AP ID or S1AP ID). We present a message flow for each connection and select the single message to see all details. We also need to check the call processing procedures (e.g. setup, release, HO, etc) and investigate all abnormal behaviors (e.g. call drops, Negative-Acknowledgment (NACKs) etc). This way we're able to check if the whole feature is working correctly.

Trace Management allows the customer to look at the mobile network on different levels, not only network level KPIs. With Trace Management a single UE, call or cell can be monitored in order to get useful information for troubleshooting or feedback about network quality and capacity.

Adam Bartkowski
R&D Manager, Feature Verification

3. Summary

Our goal is to provide our customers with the most up-to-date LTE solutions with assured SW quality. This requires constant product monitoring and testing. We realize this by focusing on different areas in E2E system level tests. We can be sure about one thing – there's nothing that we can't do in our laboratories to gain confidence in our LTE product quality. Each day we challenge ourselves by going deeper and deeper into customer's requirements through cooperating closely with architects, developers, entity testers and customer teams. Everything is done to achieve superior system testing results.

References

[1] TS 32.421 Telecommunication management; Subscriber and equipment trace; Trace concepts and requirements
[2] TS 32.422 Telecommunication management; Subscriber and equipment trace; Trace control and configuration management
[3] TS 32.423 Telecommunication management; Subscriber and equipment trace; Trace data definition and management
[4] 3GPP specifications – http://www.3gpp.org/specifications/specifications

About the author

Graduate of Wrocław University of Technology, awarded a M.Sc., Eng. in ICT in Faculty of Electronics. I'm leading a team of engineers in Global Verification organization within MBB FDD LTE business line. My team is responsible for end-to-end system tests for Nokia LTE product.

Radosław Idasiak
R&D Manager,
LTE Feature Verification

Determining the Priorities of eNodeB Software Tests

Szymon Góratowski
Integration and Verification Engineer
MBB FDD LTE

NOKIA

1. Introduction

Software testing is an inevitable part of software development process and it should be started as soon as possible. Software errors can occur at every stage of the software development process. Finding errors as fast as possible when creating software can save time, money and employees' effort. Undetected errors can lead to serious consequences in the future implementation of the project. If an error is detected in the software after selling an official build to the customer, the company providing the software loses prestige on the market. Errors detected on the customer side can also be associated with high penalties.

It is also important to separate the responsibility for testing the code between the developer and tester. For development testing (e.g. unit, component, and system testing) main objective may be to cause as many failures as possible so that defects in the software are identified and can be fixed. Testing and debugging are different. Debugging is the development activity that finds, analyzes and removes the cause of the failure [1].

2. Unique test environment

Testing software which is running on hardware is significantly different from testing websites or programs. The software is adapted to the requirements of the eNodeB hardware. Testers must be careful not to destroy the device during software installation and testing. Testers are also responsible for the correct eNodeB connection to the core network and to other network devices. It is very hard to achieve conditions similar to the customer's network in a laboratory. However, it is very important because eNodeB's behavior during the testing should be the same as in the real network.

3. Types of eNodeB's features tests

To verify a new functionality of eNodeB, testers need to design tests that cover the changed software code added during the feature implementation. Test types can be divided into functional and non-functional.

Functional tests verify the function of the code. Scenarios for a functional test can be found in code documentation. These tests check if new implemented functionality works as it was described in the feature specification. Functional testing considers the external behavior of the software. Interoperability tests, which show how the function interacts with one or more other functions, also belong to functional tests family [1].

Non-functional tests are not related to the function of the code. They include performance testing, load testing, stress testing, usability testing, maintainability testing, and reliability testing. Non-functional tests verify the external behavior of the software [1].

We can also divide the tests with regard to the scope of the new feature verification. Sometimes new features are very complex and consist of smaller functions, which can be tested separately. The division of complex feature into smaller parts facilitates the testing and simplifies analysis of the test results. After testing the components separately, test of the entire feature should be focused on the input conditions, final results, and interactions between previously tested components.

It is also worth to mention the regression tests. Regression testing is repeating tests already tested on the software after code modification. Those tests are performed to discover defects, which can be caused by the code modification [1]. Regression testing should be automated as much as possible. Automated environment setup and automated regression tests are faster than manual testing. Shorter testing time results in a lower testing cost.

Another very important type of software test is interoperability test. Implementation of new functionality in the software occasionally has impact on previously implemented functions. It is very important to check whether the implementation of a new functionality does not bring errors to interaction with cooperating functions. If there is a possibility to activate several features of eNodeB at the same time, including the one newly implemented, there is a need to verify the correct operation of all functions.

Tests can also verify the stabilization of the software code. For this purpose, stability tests are created. They check the stability of the implemented function for longer time than a single functional test. In practice, duration of the stability testing should be longer than 24 hours. Such a long testing time is required to check whether a newly implemented feature does not suspend nor crash the eNodeB. Stabilization tests also verify the capacity of the memory buffers on the eNodeB and if there is no overflow. Memory buffers should not be filled to the maximum capacity because it can lead to eNodeB's slow down or even a crash.

There are also other divisions of software tests, for example, with regard to the purpose of testing or the means used.

4. Prioritization of eNodeB tests

Software testing can provide objective, independent information about the quality of software. When testing detects a bug, the code is subsequently repaired and the overall quality of the software increases. Definition of software quality may be twofold:

1. Software functional quality – how the final software product meets the technical documentation and design objectives.
2. Software structural quality – how the final software product reflects non-behavioral requirements [2].

The purpose of testing is to eliminate bugs and errors, which appear in the software code. But we must remember that there is no possibility to verify all scenarios. That is why we need to perform prioritization of eNodeB tests.

Number of tests to execute should be determined by the level of risk (technical and business) and budget limitations. An important factor is also the time of testing. If there is enough time testers can increase the number of tests or perform tests which require extended period of time.

Unexpected situations or communication problems can cause delays in the software testing. Deficiency of time forces the tester to appropriately classify the tests, in other words, to determine which tests are critical and must be performed. It also requires smart design of test plans, so that the larger part of a new feature could be verified in one test. To reduce the number of test cases, tester should select feature parts which are the most vulnerable to errors. This requires deep knowledge and understanding of eNodeB architecture.

Test priority should depend on the complexity of implemented function. The complex logic of software is one of the most common causes of errors. Therefore complicated parts of the code should be fully tested.

There is no possibility to test all software components. Tester has to choose tests that will verify parts of the code that are sensitive to errors. The hierarchy of priorities should take into consideration:

- complexity of code elements.
- importance of software function implemented by the change.
- value of functionalities from the customer point of view.
- feedback from developers about the part of software most prone to errors.
- testing methods covering the largest number of functions.
- other parameters specific for the project.

Prioritizing is important in regression testing. By setting the test priority, tester can select important tests and transform them into regression tests. Regression tests have to be performed after every change of the code. The aim of these tests is to check if a function is working correctly after change implementation.

During testing tester ought to find the right balance between the amount of testing and software code coverage. Prioritization of tests helps to do it properly.

5. Conclusion

Developing suitable test is not easy. Prioritization can highlight code functions that are important for some reason. Importance hierarchy of test may depend on various factors, such as type, duration, and other complexities. It depends on what tester wants to achieve in the project objectives. Testing software developed for base stations is very complex and requires knowledge about the whole system and base station architecture. The eNodeB's software consists of multiple components. Therefore tests verifying proper operation of the component functions have to be adapted to the project documentation requirements.

Determining the priorities allows testers to classify components with regard to degree of modification. This technique helps in selecting important software components and designing tests for them.

It is impossible to test the whole software without specifying the depth of code infiltration. To determine the depth of software penetration by tests, tester should set priorities for all tests. Assigning test priorities helps avoiding the creation and execution of unnecessary tests, determining the importance of tests, and selecting the most important tests if the time to complete the testing is insufficient.

References

[1] "Certified Tester – Foundation Level Syllabus", international Software Testing Qualification Board.

[2] Pressman, S., Software Engineering: A Practitioner's Approach (Sixth, International ed.), McGraw-Hill Education 2005.

About the author

I am a graduate of the Faculty of Telecommunication at Electronics Department of Wrocław University of Technology. I am working as an Integration and Verification Engineer in MBB FDD LTE C-Plane department. Our team tests software developed for the new functionality of base stations.
My responsibility is to test the new functionality in eNodeB software. It is an interesting and challenging work that requires constant improvement of knowledge of LTE technology.

Szymon Góratowski
Integration and Verification Engineer
MBB FDD LTE

LTE L1 Call – the Necessary Condition for LTE Testing

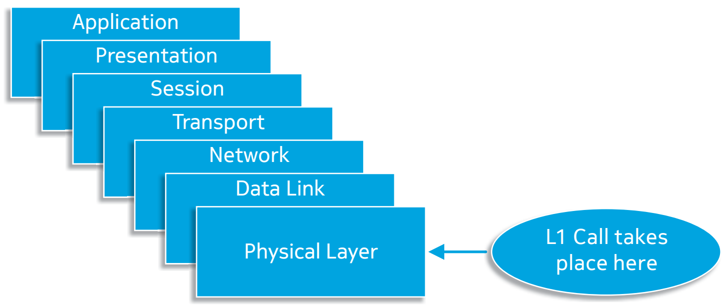
Marek Salata
Engineer, Hardware Integration
MBB Radio Frequency

Introduction

Every person who has a basic understanding of how contemporary telecommunications infrastructure works must realize that before any such system begins its commercial operation, it has to be extensively evaluated. A set of tests aims at determining whether the equipment used indeed functions as expected and conforms to both internal company norms and international standards, as well as supports interoperability with already deployed systems.

The necessity to pass restrictive tests pertains to all elements of the network; however, this paper focuses specifically on an aspect of Radio Module testing. It is one of the more frequently changing elements of the otherwise quite static network infrastructure, and, as such, it comprises a broad range of equipment to choose from and an equally vast testing ground. Radio Modules are typically located on radio masts and have antennas connected to them to generate electromagnetic signals, which ultimately give a cell its coverage. There are multiple operating bandwidths, powers, and different access technologies involved, depending on the age of infrastructure, the needs of the target market, deployment sites, preferences of operators, how densely occupied a given frequency spectrum is, expected cell coverage, and other factors. These features of radio networks are common for multiple technologies, both frequency division duplex (FDD) and time division duplex (TDD).

Figure 1 L1 Call in relation to layers of the ISO OSI model. [1]



The fundamental idea behind testing of a given radio module is to determine if it works properly on the physical layer of the OSI model (see Figure 1). After all, there is no point in trying to transfer and receive packets of tangible data when elements responsible for managing the air interface are not working according to the design. We need to ensure that the physical properties of an electromagnetic signal generated at the radio antenna are exactly what the product specification predicts.

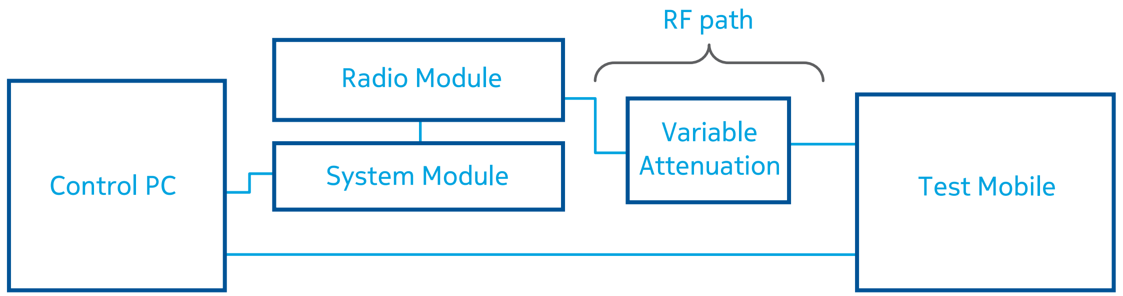
On the other hand, an equally important task is for the radio module to properly decode a signal it has received from the outside. 3rd Generation Partnership Project (3GPP) specifications for long-term evolution (LTE) predict several different test models used to emulate particular behavior of the equipment in question, considering various parameters we intend to measure. Depending on the needs and capabilities of devices, downlink and uplink signals may use variable modulation and coding schemes, different number of resource blocks, and many other features which adapt a system to a particular working environment. In order to understand the idea behind LTE test models, and subsequently the L1 Call, one also has to know what orthogonal frequency division multiple access (OFDMA) and single-carrier frequency division multiple access (SC-FDMA) are. These two modes of LTE access are used for downlink and uplink respectively. Generally, for radio measurements of downlink (transmission from the base station to the user equipment; further referred to as DL), a signal analyzer is required and Evolved Test Models (E-TMs) are used. Similarly, for uplink (transmission from the user equipment to the base station; further referred to as UL), we need to use a signal generator and fixed reference channels (FRCs).

Thus, what is a test model? The simplest explanation is that it is a set of channels where known data is encoded and transmitted in one of many predefined ways. Depending on what is needed, test models are used for specific testing such as measuring adjacent channel leakage ratio (ACLR), error vector magnitude (EVM), block error rate (BLER), constellations, signal-to-noise ratio (SNR), and so on. However, both DL and UL are measured apart from each other, and in this separate radio frequency (RF) context each is accordingly evaluated.

Aside from cases for the two situations – uplink and downlink – mentioned in the previous paragraph, one of the most important tests is that of a simultaneous transmission in opposite directions. This is commonly known in Nokia's technical jargon as an L1 Call; "L1" means that it takes place on the first layer of the OSI model (see Figure 1), the physical layer, while "Call" means that both sides of the connecting equipment transmit and receive signals at the same time. It is important to mention test models first because the L1 Call shares many of their features but also adds certain nuances. In very general terms, the L1 Call is somewhat similar to running both UL and DL test models at the same time, using the same RF path.

This is what has been described in this paper: what the L1 Call measurement setup looks like, how its various elements are connected, what equipment is used, and finally how the tests proper are carried out. The purpose of the L1 Call is to prove that the physical layer transmission in opposite directions can be established and maintained in preparation for higher OSI layers, where more complex data processing takes place. The main parameters that are evaluated are the block error rates and throughput for both uplink and downlink. Additionally, features such as signal modulation and constellation are checked.

Figure 2 Measurement setup.



1. Description of the measurement setup

What exactly is the L1 Call in practice? We know it is a test used to determine the ability of a radio module to simultaneously transmit and receive signals, but what does the measurement setup look like? Figure 2 is meant to shed some light on these questions.

A Control PC is connected to both the System Module and Test Mobile through Ethernet cables. The System Module is connected to the Radio Module via optical fibers or electrical links. The Radio Module is connected to the Test Mobile via an RF cable (more than one in case of multiple inputs multiple outputs (MIMO), which will not be discussed here) with adjustable attenuation, which forms the radio frequency path.

In the scope of the L1 Call, the Radio Module is called the Device Under Test (DUT), also referred to simply as RM. It is in the middle of the L1 Call setup because its antennas transmit and receive signals that are being measured on both ends. Instead of an air interface, we connect the antenna outputs directly to the user equipment emulator (in our case, it is the Test Mobile device) through an RF cable. The reason for this is that we wish to eliminate any potential sources of interference and instead focus on the physical capability of the hardware to emit a proper electromagnetic signal and receive one in return.

A System Module (SM) is an equivalent of a Base Station (BTS) and the closest thing to a core network in the L1 Call setup. This is where we in fact decode and analyze the signal that the RM has received, but we also generate one to be transmitted by the RM. The SM is a box with several programmable processors, each capable of performing separate tasks. We connect to this entity with our control PC over Ethernet. From there we can send certain instructions that will also reach the RM connected to the SM. There may be situations where the two devices are far away from each other, connected through up to six pairs of optical fiber strands dozens of kilometers long. Those are separate cases, however, since in our standard L1 Call measurements, we use shorter lengths to simplify the process.

A Test Mobile is a user equipment emulator that allows us to simulate functionality of different categories of devices (depending on a license) that would normally interact with the BTS in a typical working environment. It offers a range of operational frequencies between 400 MHz and 4000 MHz, sufficient for all bands used by present-day LTE network operators. Some types of Test Mobiles provide diagnostic and monitoring software that allows testers to analyze the user equipment (UE) side of the radio path. A Test Mobile is controlled through scripts, which give testers certain flexibility in how we wish to carry out our measurements, including changing various parameters that govern data transfer rates, modulation and coding schemes, block error rates, and a number of other features; thus, it is an exceedingly versatile device.

An RF path is typically a concentric cable of known parameters, approximately 2 meters long, with a variable attenuator in the middle. One end is attached to the RM output, the other to an antenna port on the Test Mobile.

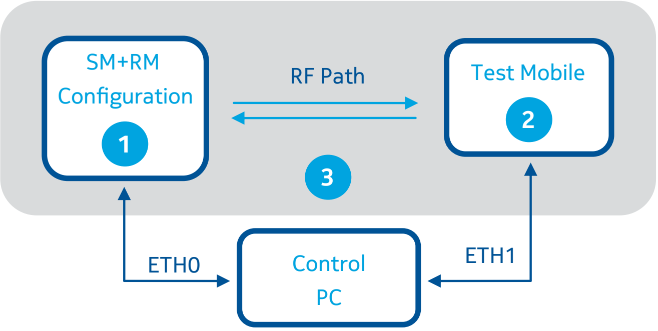
A Control PC is, for all intents and purposes, a standard desktop machine operating Nokia scripting software through which the SM and RM are configured. A radio link between the BTS and Test Mobile is established from here. The PC is connected to the devices on both ends of the radio path via Ethernet cables and is our command and control center from which we carry out our tests. This is where we can view the results and collect logging data for further analysis.

As for the software used, the Test Mobile's manufacturer provides a Windows graphic user interface (GUI) application for the UE emulator, which we use to carry out our measurements and monitor the transmission. Its underlying system can in turn be controlled remotely through another program which uses a scripting language and is the software mainframe behind everything we do in our laboratories. It facilitates connections to the SM, RM and Test Mobile, collects information and logging data from all devices, and sends control messages to System and Radio Modules. It encompasses multiple technologies; it detects and configures all types of hardware we may want to use, down to minute differences in types of processors, firmware versions, number of DUTs, the sort of small formfactor pluggables (SFPs – optical-electrical signal converters) we are using, and a great many other things.

2. L1 Call procedure

The L1 Call is essentially a straightforward and logical procedure. Figure 3 presents a block diagram of each step that has to be taken to finalize a call; each step has been described below.

Figure 3 Basic steps that lead to establishing L1 Call.



1) Hardware Configuration: it is a process of setting up all the components of SM and RM to be ready for signal processing. Then, a cell setup both for downlink and uplink takes place; reference and synchronization signals start being transmitted, and the RM activates its carriers. The end result is that the RM is generating an electromagnetic signal at its antenna port (or ports) and is also set to receive incoming signals. We only have one user that the whole physical downlink shared channel (PDSCH) is allocated to. Predefined data in downlink is transmitted in an organized manner, much like what happens during the DL test model tests. UL is prepared to receive data in return. This is also the only point of difference between TDD and FDD in the L1 Call; for TDD, we have to accommodate sub-frames separated in time instead of frequency.
2) Test Mobile configuration: what is left is activating the UE emulator and connecting with the cell that has been set up in the previous step. Radio output/input of both the RM and Test Mobile are connected through an RF cable, so the air interface is also ready. The Test Mobile undergoes a similarly complex process of setting up its own cell (analogous to what UEs normally do when powered up) and synchronizes with our BTS. This leads us to the final point.
3) Measurements: when synchronization is complete, the L1 Call has been established. Test Mobile and SM+RM are sending and receiving data to and from each direction, which can be verified using applicable monitoring software. Figures 4 and 5 present an example of the L1 Call for 10 MHz bandwidth and frequencies of 2,625 GHz for DL and 2,505 GHz for UL.

A Nokia application (see Figure 4) allows us to verify uplink transmission. Several parameters are checked versus expected values: signal-to-interference-plus-noise Ratio (SINR) and BLER. The constellation has to be clear and discernible, SINR has to be above zero, and BLER has to be equal to zero; everything during a measurement period of one minute. If these conditions are met, we know that UL works as expected. The signal has been sent from the UE, through the RF path, received and decoded at the RM, and finally sent to the SM over an optical fiber. The purpose of the L1 Call was to test the Radio Module's ability to send and receive. We now see that the signal has passed through every part of that network element, from the antenna port all the way to the interface between the RM and SM, and the rest of our BTS was able to further process it. We can see a very strong signal (SINR around 30.4) transmitted with a 16QAM modulation, clear constellation and no errors.

A sample measurement using Test Mobile's interface is shown in Figures 5 and 6. It is the same L1 Call test scenario as presented in Figure 4, but this time we analyze downlink. Our signal, with a 64QAM modulation, (see Figure 5) shows a clear constellation and no transmission errors. This digital signal has been generated in the SM. Then, it has been sent to the RM, which has converted it to an analog signal, and transmitted it through the antenna over the RF path to our UE emulator. There, it has been received and analyzed also during a one-minute measurement period.

Figure 4 Sample UL measurement using an LTE Browser – a Nokia application which gives us an insight into what is happening on the SM.

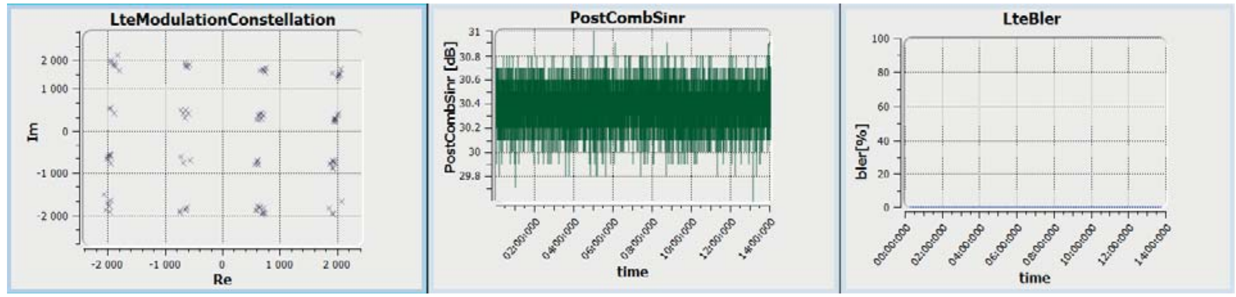
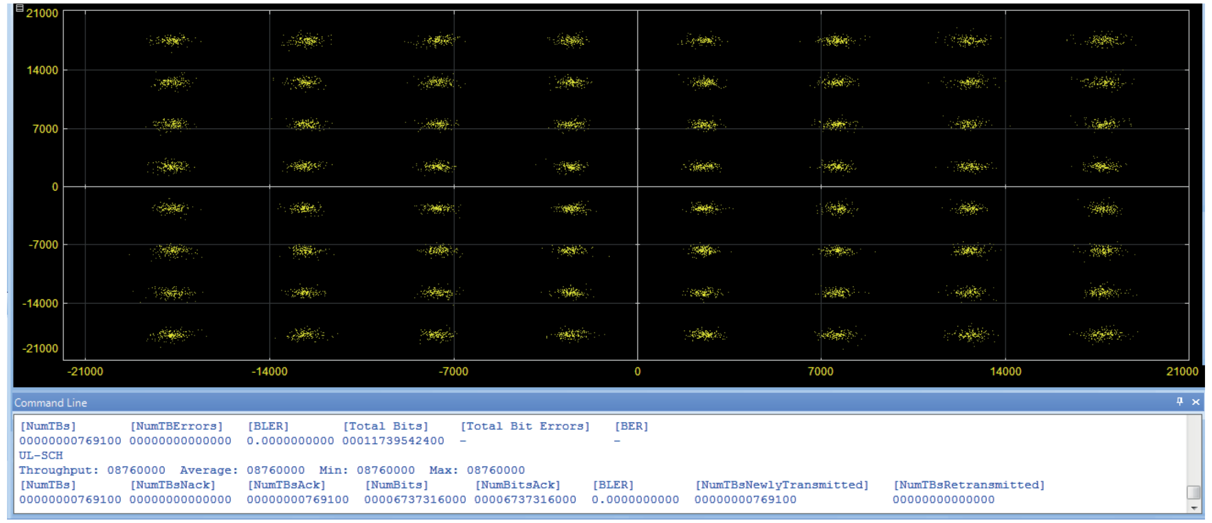
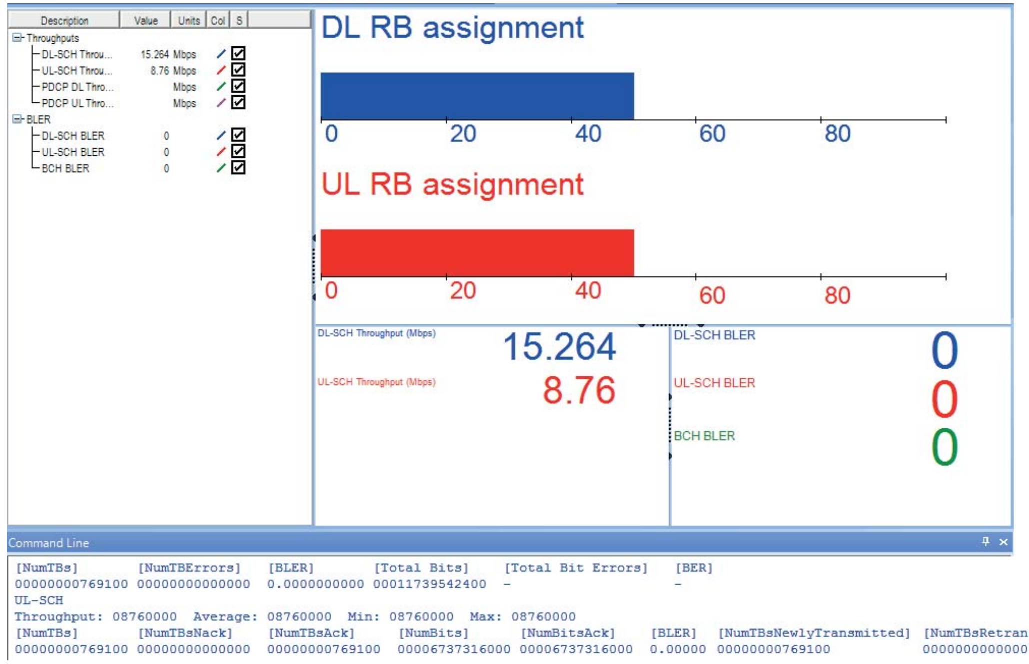


Figure 5 Summary of PDSCH and DL signal.



```
[NumTBs]        [NumTBErrors]   [BLER]        [Total Bits]    [Total Bit Errors]    [BER]
00000000769100 00000000000000  0.0000000000 00011739542400  -                     -
UL-SCH
Throughput: 08760000  Average: 08760000  Min: 08760000  Max: 08760000
[NumTBs]        [NumTBsNack]    [NumTBsAck]     [NumBits]       [NumBitsAck]    [BLER]         [NumTBsNewlyTransmitted]    [NumTBsRetransmitted]
00000000769100 00000000000000  00000000769100  00006737316000 00006737316000  0.0000000000  00000000769100              00000000000000
```

Figure 6 Throughput and BLER for both uplink and downlink, including resource block assignment.

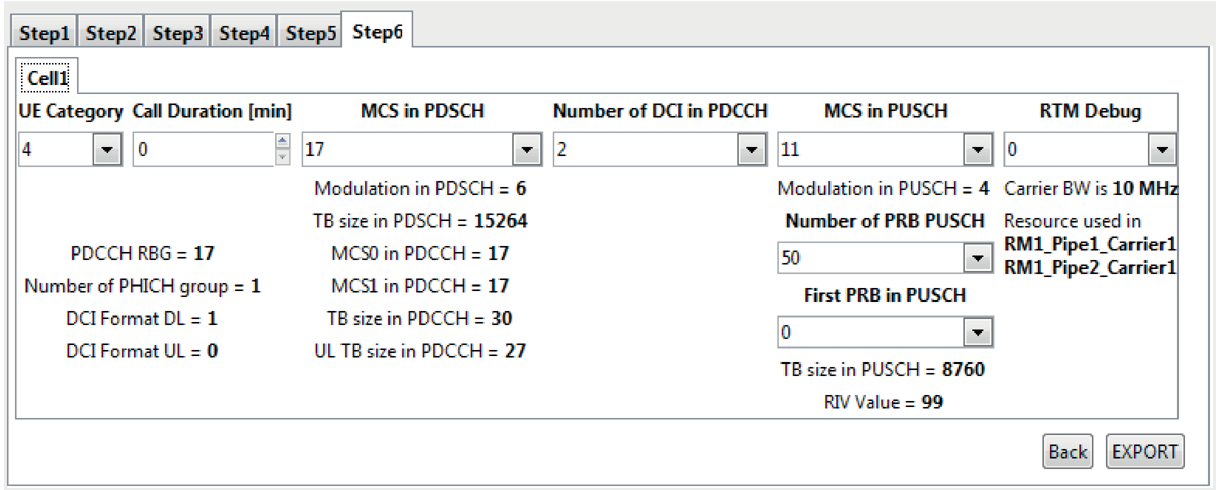


```
⊟ Throughputs
  ├ DL-SCH Throu...   15.264 Mbps  ☑
  ├ UL-SCH Throu...     8.76 Mbps  ☑
  ├ PDCP DL Thro...          Mbps  ☑
  └ PDCP UL Thro...          Mbps  ☑
⊟ BLER
  ├ DL-SCH BLER            0       ☑
  ├ UL-SCH BLER            0       ☑
  └ BCH BLER               0       ☑
```

DL-SCH Throughput (Mbps) 15.264
UL-SCH Throughput (Mbps) 8.76
DL-SCH BLER 0
UL-SCH BLER 0
BCH BLER 0

```
[NumTBs]        [NumTBErrors]   [BLER]        [Total Bits]    [Total Bit Errors]    [BER]
00000000769100 00000000000000  0.0000000000 00011739542400  -                     -
UL-SCH
Throughput: 08760000  Average: 08760000  Min: 08760000  Max: 08760000
[NumTBs]        [NumTBsNack]    [NumTBsAck]     [NumBits]       [NumBitsAck]    [BLER]   [NumTBsNewlyTransmitted]  [NumTBsRetransmitted]
00000000769100 00000000000000  00000000769100  00006737316000 00006737316000  0.00000  00000000769100            00000000000000
```

Figure 7 One of the steps in a configuration tool.



UE Category: 4
Call Duration [min]: 0
MCS in PDSCH: 17
Number of DCI in PDCCH: 2
MCS in PUSCH: 11
RTM Debug: 0

PDCCH RBG = 17
Number of PHICH group = 1
DCI Format DL = 1
DCI Format UL = 0

Modulation in PDSCH = 6
TB size in PDSCH = 15264
MCS0 in PDCCH = 17
MCS1 in PDCCH = 17
TB size in PDCCH = 30
UL TB size in PDCCH = 27

Modulation in PUSCH = 4
Number of PRB PUSCH: 50
First PRB in PUSCH: 0
TB size in PUSCH = 8760
RIV Value = 99

Carrier BW is 10 MHz
Resource used in
RM1_Pipe1_Carrier1
RM1_Pipe2_Carrier1

At this point, the test has been successfully carried out. We have a clear indication of that on the opposite ends of the system; in the UE and SM. There are no errors, expected throughput has been achieved, all other attributes of the transmission are correct. DL throughput is higher than UL because of different values of the modulation and coding scheme (MCS) used for each. This and several other important parameters used in the whole configuration process are presented in Figure 7. They need to be defined at the very beginning of the L1 Call. Those parameters govern exactly how the test is carried out, what throughputs can be expected, and if there is a chance of errors occurring during transmission. Figure 7 presents one of the configuration steps, where those attributes are set for a 10-MHz bandwidth.

Figure 7 shows a Nokia application used to set up all of the parameters required to configure the System Module, Radio Module, and Test Mobile emulator in preparation for the execution of the L1 Call test case.

The following is a description of parameters in Figure 7:

- MCS: It is one of the crucial parameters that govern several characteristics of our signal such as the type of modulation used: quadrature phase shift keying (QPSK), 16 quadrature amplitude modulation (QAM), or 64QAM, as well as the number of transport blocks and organization of resources in PDSCH and physical uplink shared channel (PUSCH). Ultimately, it has a major impact on the achieved throughput. Modulation in PDSCH and PUSCH in fact stands for a modulation order (refer to Table 7.1.7.1-1 in [4]) and is dependent on the MCS index.
- Resource blocks: RBs represent system capacity in uplink and scale with bandwidth used. We can manually assign a non-standard number of physical resource blocks and observe how the system reacts to that, but most of the time, the maximum number permitted for a given bandwidth is used (5, 10, 15, and 20 MHz use 25, 50, 75, and 100 physical resource blocks (PRBs) respectively; 1.4 MHz and 3 MHz can also be used). The number of resource blocks utilized is also defined in the 3GPP specification.
- UL transport block (TB) and TB size in PDCCH: Once again, it is a 3GPP-specification-defined value, and thus it cannot be directly modified by users; Transport Block values are calculated based on other parameters. The same is true for several other variables presented in Figure 6, such as the Resource Block Group (RGB), Downlink Control Information (DCI) format, or Resource Indication Value (RIV).
- UE category can also be modified in case we wish to emulate a specific type of device and see how the DUT reacts to that.
- Note how transport block sizes for PDSCH and PUSCH directly translate to the achieved throughputs; 15.264 Mbps in DL and 8.76 Mbps in UL. Detailed calculations of expected throughputs and formulas used can be found in 3GPP specifications.

3. Conclusion

The L1 Call may not be a particularly complicated or difficult test case, but it is certainly the very first one where the Radio Module does what it was designed to do – it receives and transmits data at the same time. The test allowed us to determine that the DUT was able to conduct transmission without errors. We can see the actual transmission speed, which is a more familiar value that gives us an indication of the system's capacity and scalability. However, due to the fact that we are restricted to the physical layer, many simplifications have to be taken into consideration.

Firstly, we do not divide the PDSCH between users, which is what would happen in a typical commercial implementation. This eliminates a significant element of complexity, where we do not have to worry about arranging resources in a unique way for each user. Secondly, the Channel Quality Indicator (CQI) is a redundant concept for us because we utilize perfect conditions throughout the entirety of our radio path. We even force the hybrid automatic repeat request (HARQ) not to use any retransmission via a special scripting in SM software. Because of that, any problems with BLERs we typically experience are likely as a result of a faulty connector, a synchronization cable loose in its socket, or an occasional fault in software.

Our equivalent of a BTS does not have to dynamically assign MCS and PRB values, depending on the CQI factor, which would normally take place as users continuously change their geographic location in relation to the BTS and face shifting interferences. Thus, we can emulate a best case scenario. It is important because the job of our team is not to stress-test a brand new piece of equipment but rather to ensure that a Radio Module which just got off the assembly line indeed works and does not experience any hardware faults during a normal operational cycle. When the L1 Call is done, it is a signal for other teams to start performing more advanced tests, optimizing the software and fine-tuning a great number of other features. It could be said that in the scope of our work, we explore some previously uncharted territories and pave the way for the rest of the Nokia Corporation's R&D branch to build upon that foundation.

References

[1] ISO standard 7498-1:1994
[2] Rumnay M., "LTE and the Evolution to 4G Wireless: Design and Measurement Challenges," Agilent Technologies, 2 edition, March 2013.
[3] 3GPP Test Specification 36 141 V12.7.0
[4] 3GPP Test Specification 36 213 V12.5.0
[5] 3GPP Test Specification 36 101 V12.7.0

About the author

I am a graduate of Electronics and Telecommunications faculty of Wrocław University of Technology. I work as an engineer in the MBB RF Hardware Integration and Verification team. Our branch of Nokia's R&D is concerned with hardware tests of prototype radio and system modules and preparation of configuration scripts for other teams. Our tasks revolve around green-lighting new equipment as it emerges from Nokia's assembly lines, and we are at the forefront of hardware and software fault detection and evaluation.

Marek Salata
Engineer, Hardware Integration
MBB Radio Frequency

Digital Linearization of RF Transmitters

Krzysztof Kościuszkiewicz
Software Architect
MBB Radio Frequency

Karol Sydor
Software Development Engineer
MBB Radio Frequency

1. Introduction to RF transmitters

Energy efficiency is one of the key design goals in RF telecommunications systems, both in handsets and base stations. In handsets, it directly affects the battery life and hence the end-user experience. On the BTS end, higher energy efficiency is coupled with equipment size, cooling methods, achievable transmitter power, and operating costs by the network operator. In fact, up to 80% of energy consumption in the BTS can be attributed to the RF transmitter. Therefore, even small improvements in the transmitter efficiency will impact energy costs, achievable throughput/cell size, and other related aspects. To put things in perspective – 6TX 40 W Nokia Flexi Radio consumes ca. 16.5 kWh per day with rated ETSI 102 706 traffic load model. With transmitter efficiency at 38%, even a 1 percentage point increase translates into a 2% decrease in energy costs across the whole network.

Modern cellular technologies are based on wideband signals with a high dynamic range. This, in turn, requires the usage of highly linear transmitters to amplify the signal without significant degradation in its quality. It is in the transmitter design where linearity requirements and efficiency of the BTS are determined. As it is often encountered in engineering, one goal opposes the other and it is a real challenge to design an efficient and linear RF amplifier.

Figure 1 illustrates the typical amplitude and phase nonlinearities of a 40 W Doherty Power Amplifier (PA) [1]. With a 6.7 dB peak to average rate (PAR), the PA needs to handle up to 187W signal peaks to transmit with an average power of 40 W. The linearity of the PA can be easily improved by shifting its operating point to the left. This is equivalent to overdesigning the device – 6 dB back-off in the operating point is equivalent to the reduction of the average PA power from 40 W to only 10 W, and a sharp drop in efficiency – from 38% to 23% in this example.

2. Analog linearization systems

One strategy devised to have both high linearity and high efficiency was to implement a system where transmitter nonlinearity is estimated and corrected automatically. Let us briefly review analog linearization solutions implemented in the past.

Figure 1 Single tone gain and phase PA characteristics.

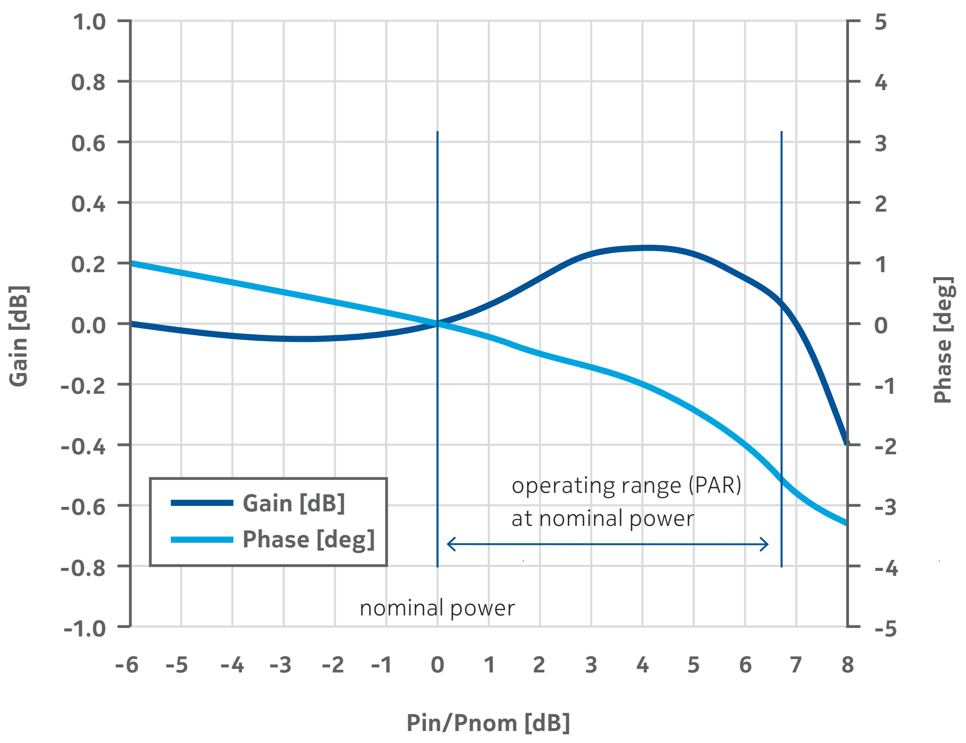


Figure 2 Analog closed loop linearization system.

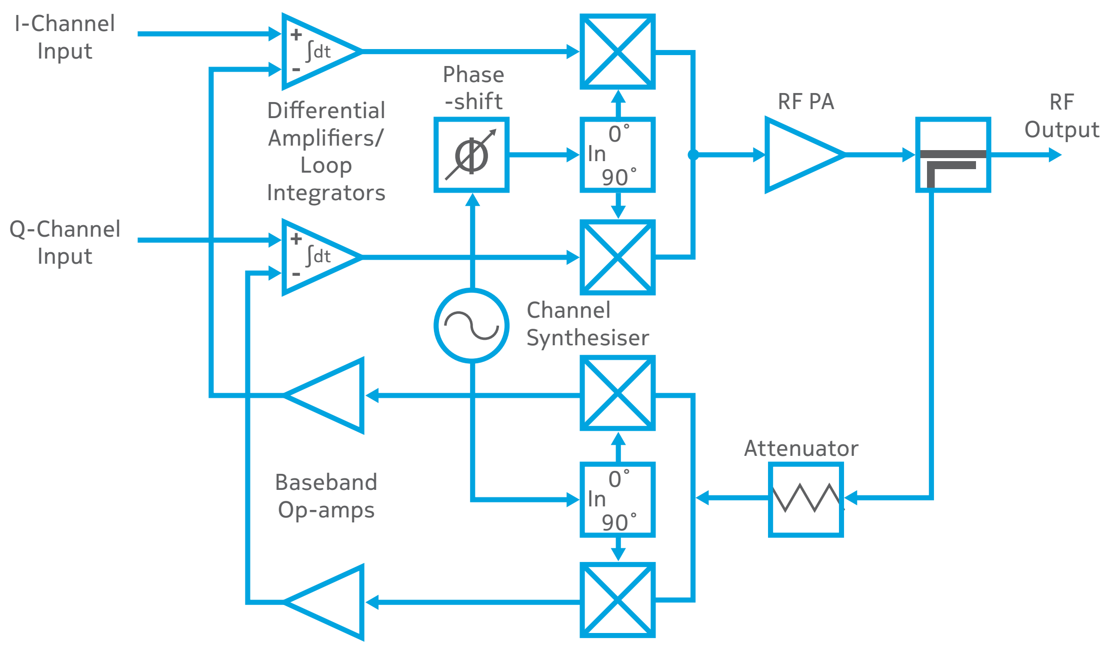


Figure 3 Analog feedforward linearization system.

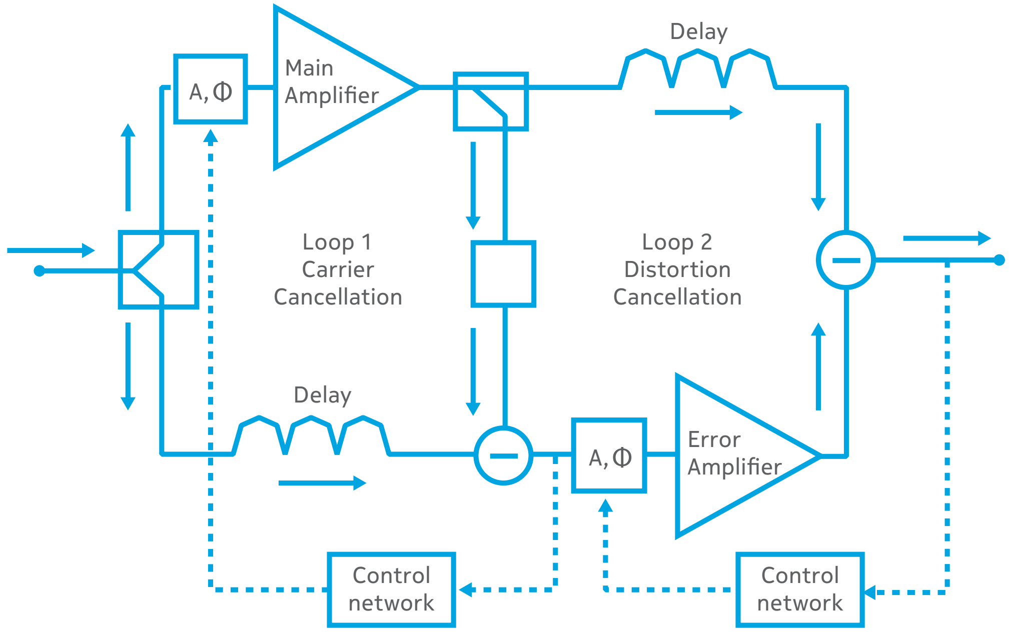
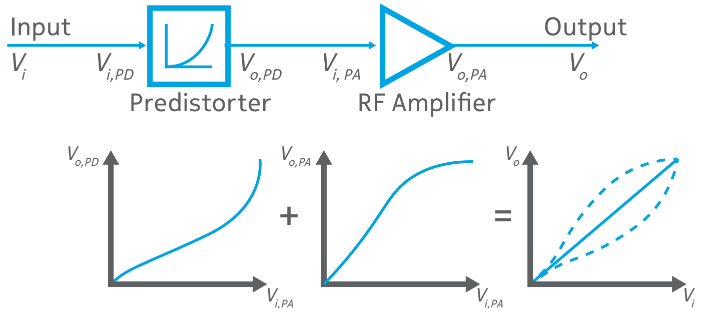


Figure 4 Principle of linearization with digital predistortion.



In the first approaches (Figure 2), nonlinearity was corrected directly via the closed analog loop feedback system where corrections were estimated and applied directly to the in phase and quadrature signals before the upconversion. A linearity improvement up to 35 dB was achievable in similar systems, but the linearization bandwidth was limited up to about 1 MHz by the feedback bandwidth and time response characteristics.

The Design of a so-called "feedforward" linearization is illustrated in Figure 3. In theory, they could achieve a high bandwidth and up to a 20–30 dB ACLR improvement. But, it should be noted that both the error amplifier and the top right delay component must operate at a nominal RF power of the unit. This resulted in a high power dissipation and physically large components required to implement such solutions, limiting the achievable efficiency to 15–20%.

3. Digital predistortion

Limitations of the analog solutions could be side-stepped with a digital realization of the linearization systems. This became practical with high bandwidth analog-to-digital converters (ADC) used in feedback loops and high speed digital system processing (DSP) realizations. Digital Predistortion (DPD) is the most common technique of PA linearization in the digital domain. Its principle has been illustrated in Figure 4.

The linearization is achieved by the composition of the PA and the Predistorter characteristics. In general, DPD consists of two main blocks – the predistorter and the identification block. Predistorter applies the inversed behavioral model of the RF Amplifier to the downlink signal samples. It can be realized by a DSP processor, or a dedicated HW block. The identification block is responsible for the estimation of the behavioral PA model parameters. This block requires hardware support for signal capturing, and the remaining part can be realized in different ways, depending on the used architecture.

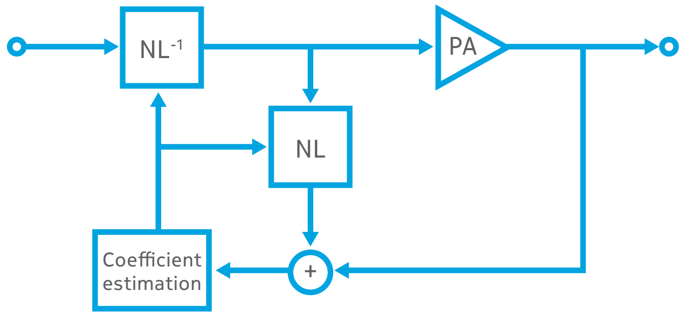
In the most common approach, the identification process is based on mathematical modeling. Using the Volterra theory [2] it is possible to describe non-linear behavior of the PA by using a linear combination of coefficients, and non-linear functions called the Volterra kernels. Software processing can extract the non-linear Volterra model in a direct manner by using the least squares linear system identification algorithm [2].

There are two main realizations of the above described predistortion approach. Depending on the requirements and available resources, a suitable solution can be selected. The first approach is the Direct Learning Approach (DLA), with its system diagram depicted in Figure 5 below.

In this approach, the nonlinear PA model (NL) is identified by estimating the coefficients of Volterra-kernels. Basing on that, the indirect model (NL) is created and applied to the transmit signal. The main benefit of this approach is that the bandwidth of the PA model is dependent only on the sampling rate in the transmit pipe, and not the feedback path. Undersampled, real-only feedback path can be used and this reduces the overall system cost. The main disadvantage of this method is that a direct PA model is identified, while the inverse model is needed for predistortion. This implies the need for inverting the numerically complex model.

There are two possible solutions for this issue. One is numerically calculated inversion. Since the PA-model consists of higher order non linear functions and high levels of memory, numerical inversion is a complex procedure requiring a high amount of processing power. The second solution is to use the iterative approach. This method can be accelerated by using a dedicated hardware solution, which supports several iteration processing. This eliminates the need for software based inversion and reduces the processing time.

Figure 5 System diagram of Direct Learning Approach.



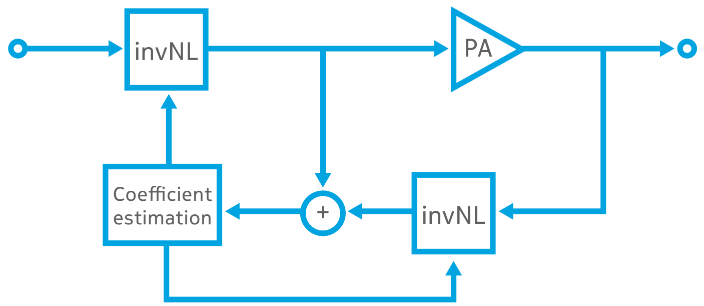
The iterative inversion approach is using the fixed point method to approximate the inverted PA model iteratively, by using the unity gain as the fixed point. This requires the PA model to be normalized in a way that ensures that most of signal samples will pass through the predistorter without gain change. The PA model is normalized by the software by numerical factor based on its raw gain for nominal power. The fixed point method ensures that after several iterations (desired performance can be achieved after 4 iterations) the error of the approximation converges to zero. This means that the iteratively approximated inverse model behaves as the desired inverse PA model.

One of the disadvantages of the fixed point method is the decreasing accuracy of inversion for parts of the PA model which are further away from the nominal operating point on the gain and phase. With linear convergence further improvements would require costly additions of more iterations in the accelerator. This can be limited by adding Convergence Accelerator to the predistorter processing path – a non linear function which extracts the gain curve from the remaining part of the PA model. This makes the iteratively inversed part of the PA model gain more flat in power domain, and enables to cover even more gain correction in the predistorter and still keep close to the fixed point.

This method is widely used in currently produced Nokia Radio Modules, known as iiDapd (first release of iiDapd did not contain a convergence accelerator).

Another approach to the DPD is the Indirect Learning Approach (ILA) – **Figure 6**.

Figure 6 System diagram of the Indirect Learning Approach.



In this approach, a so-called “training” model is introduced (invNL). The parameters of the training model converge with the inverse nonlinear system, and can be used in identification and in the predistorter in the same form (without any need of recalculation). This system does not need PA model inversion, and provides a similar performance as the DLA approach. The disadvantage of this method is that the bandwidth of the system is dependent on the sampling rate of the feedback path. This requires complex, high speed feedback and therefore results in a higher total cost of the system. This approach will be used in next generation Nokia Radio Modules because it requires a simpler predistorter (iterative inversion is not used, so HW capable of performing single iteration is enough to fulfill the signal processing needs).

The Nokia DPD solution is using hardware solutions with software based model calculation and validation. The parts which require real-time interaction with downlink signals are handled by the hardware. This includes the predistorter, and the parameter estimation block with a capture unit, identification block, and correlator. Software is responsible for the Volterra kernels coefficient calculation, the creation of the PA model, and validation. This approach is used in all in-house Nokia HW with integrated DPD blocks.

4. A history of DPD in Nokia

From a historical perspective, DPD has evolved altogether with other parts of Nokia BTS systems. The first digital predistortion deployed in Nokia BTS was based on neural networks. The predistorter was build up from two parallel networks, one for amplitude and one for phase compensation. Networks were initially trained to provide a zero function (no signal modification). The neural network training algorithm was minimizing the residual error between the original and feedback signal. This solution performed very well when compared with analog methods like feedforward used in similar systems at that time. However, growing demands for higher bandwidth resulted in this approach not being suitable as it could be used for systems capable of handling single WCDMA carriers.

A performance comparison of different methods showed that models with a function based predistorter with an inversed PA model perform better. In the next generation DPD, the predistorter was able to apply several polynomial functions over state variables composed from amplitude, phase, and their derivatives. The coefficients were selected based on different schemes. One scheme was based on signal parameter measurements. Several predistortion quality criteria were calculated (ACLR, EVM etc.) and coefficients were selected based on some predefined relationship between certain measurement and coefficient values. Other schemes were based on FFT measurements and coefficient scanning to obtain optimum values. In general, the system was too complex, required factory calibration, and in field additional parameter tuning and storage. This solution gave the performance needed to linearize 2 WCDMA carriers with ACLR up to -60dBc.

The next step in DPD was the first iiDapd system, designed to fit into multiradio supporting 20MHz of bandwidth and support of WCDMA or LTE carriers. It was a 4 iteration iiDapd. It was a huge step forward compared to previous solutions – the algorithm was converging from a constant starting point, so the factory calibration was no longer needed. Also, since the identification block was hardware based, the software was simplified, eliminating the need of quality indicator measurements. The first version of iiDapd was a more cost-effective, faster, and stable solution with a performance at about -73 dBc for WCDMA carriers.

The iiDapd system was improved over time. At first, the PA model memory modeling capabilities have been improved by making the predistorter able to handle a greater number of non-linear functions. Together with this feature, GSM support was introduced with its large number of previously unknown problems related to dynamic signals. Later on, the Convergence Accelerator block was introduced, giving the ability to support up to a 60 MHz bandwidth.

Newer solutions using iiDapd have focused more on cost reduction. The number of iterations was reduced from 4 to 3. The whole system has been fitted into a single chip together with a dedicated processor core, 2 linearization paths, and one shared feedback.

The currently developed projects will be the first to use ILA instead of iiDapd. The reason for this is further cost reduction. The ILA method needs a complex full speed feedback ADC, but reduces the predistorter complexity – single iteration of iiDapd predistorter is enough for the ILA algorithm to provide similar performance. Together with a verified concept of shared feedback, it allows to support more linearization paths with a single chip.

The target for next generation solutions is to support more linearization paths within one chip, less power consumption, less external components, and higher bandwidth. For comparison, one of the first systems with digital predistortion required a dedicated DPD chip to linearize 1 or 2 WCDMA carriers. The currently available radio modules contain 2 DPD paths in a single chip together with a dedicated ARM core for DPD routines and only 1 feedback path, providing 80MHz of bandwidth capable to run LTE, GSM, and WCDMA carriers.

References

[1] A new high efficiency power amplifier for modulated waves. *William H. Doherty*, Proceedings of the Institute of Radio Engineers, Vol. 24, No. 9, 1163-1182, Sept. 1936

[2] Theory of Functionals and of Integrals and Integro-Differential Equations. *Vito Volterra*. New York: Dover Publications, 1959.

[3] Digital predistortion of power amplifiers using look-up table method with memory effects for LTE wireless systems. *Ruchi Singla and Sanjay Sharma* EURASIP Journal on Wireless Communications and Networking 2012:330 http://jwcn.eurasipjournals.com/content/2012/1/330

About the authors

My journey in Nokia began as an Embedded Software Developer for the RF module where I became part of the team responsible for development of the iiDAPD SW. Today my daily work revolves around architecture and design of the software for next generation radio modules. Together with the specification team we try to address the challenges of future BTS solutions, including Cloud concepts, active antenna systems, indoor pico BTS, and 5G technology.

Krzysztof Kościuszkiewicz
Software Architect
MBB Radio Frequency

I work as a Software Development Engineer in MBB RF RFSW team. In our organization we develop software for Nokia radio modules. In our daily work we are challenged by the encounter of analog radio elements and high speed digital processing. Constant evolution of the systems and the search for better solutions is the driving force behind our professional development. In our work technology is not only a tool, but also a goal.

Karol Sydor
Software Development Engineer
MBB Radio Frequency

Professional Software Development

Professional Software Development

C++17 – the Upcoming Standard

Sławomir Zborowski
Engineer, Software Development C++
MBB Single RAN

NOKIA

1. Intro

Some say that C++ is going to die in the coming years. Nothing further from the truth! After revolutionary changes in C++11 and minor improvements in C++14 the language is heading towards the next major release – C++17.

It is the middle of 2015 when this article is being written but it is already known that C++17 will be a similar game changer as C++11 once was. This time, besides long-awaited features like concepts and ranges, C++ is going to catch up with competing languages in terms of standard library completeness.

The number of proposals submitted to be included in C++17 is high. They include a variety of features like parallel execution or compile time reflection. This article focuses only on selected language proposals, which are likely to be included in C++17 and will provide powerful tools to C++ programmers. However, since the author is not clairvoyant, please bear in mind that the actual C++17 may look slightly or even completely different than the one presented here.

2. Concepts

C++ templates often operate on types that should meet some requirements. Good example is a template function fillMemory that accepts type T and fills the memory region it occupies with some value. This function should obviously check input type's traits – the most important is whether the type is trivially copyable (e.g. POD object). If it is not, bad things can happen (e.g. virtual dispatch table can be overridden with garbage). To check whether the type is POD or not, one can simply use template std::is_pod, which is shipped with type_traits library. However, not every template is that simple. In case of complex template functions and classes there are a lot of checks like this to be performed. Unfortunately, there is no readable way to specify type requirements. Also, what is more important to library end-users, in case of type not fulfilling some requirements a compiler error can look like a damaged XML document. Essentially, concepts are meant to address these two issues.

With concepts, the library writer will be able to easily specify requirements for a particular type and the compiler will be able to produce a clear error message if the user makes a mistake.

Listing 1 illustrates what the process of creating a simple concept and using it will probably look like. Of course concepts offer greater power for real-world libraries. It is highly probable that STL will be the first one to benefit from concepts. Unfortunately, real-world examples would be too lengthy to be included here.

Listing 1 Example depicting how simple usage of C++ concept could look like

```
template<typename T>
constexpr bool Addable()
{
    return requires (T a, T b) {
         // Result of expression { a + b } have to be of type T.
        {a + b} -> T;
    };
}

// Note there is no typename/class but Addable directly.
template <Addable T>
T add(T a, T b)
{
    return a + b;
}
```

3. Ranges

STL library can be viewed as a group of sub-libraries. Besides the likes of regex, threads, or atomics, there are ones that were part of STL from the very beginning: algorithms, containers, and iterators, respectively. Taking only these three into account, it is possible to treat iterators as a glue between algorithms and containers. C++ programmers are used to this design, but they often complain that it leads to cumbersome and inconvenient usage in some cases. That was one of the reasons for which a new approach was proposed – ranges, in a paper N4382 [4].

Conceptually, ranges can be viewed as a pair of iterators, but implementations can be based on length to provide better performance. It is possible to extract begin and end iterators from a range. **Listing 2** shows the simplicity achieved with ranges library.

Listing 2 Example usage of ranges library. [5]

```
// Traditional way
std::vector v { /* ... */ };
std::sort(std::begin(v), std::end(v));

// Ranges-based way
std::vector v { /* ... */ };
ranges::sort(v);
```

The range library is going to provide overloads for all standard algorithms for convenience. Another important benefit of using ranges is that it reduces the risk of making a mistake like, for example, passing two iterators initialized with different containers into one algorithm expecting two iterators for the same container.

Yet another useful novelty that ranges library bring is views. They can be viewed as ranges that represent a lightweight view of an underlying sequence. This kind of view is cheap to copy and does not take ownership of elements from the container. With views it is possible to easily pipeline mutations and apply actions in a way that resembles functional programming. The following example code fragments from ranges' library documentation illustrate the power behind views concept.

Listing 3 Example usage of views from ranges library. [5]

```
// Example: convert vector of integers to a view with
// even numbers represented as strings.
std::vector<int> vi{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
using namespace ranges;
auto rng = vi | view::remove_if([](int i){return i % 2 == 1;})
              | view::transform([](int i){return std::to_string(i);});
// rng == {"2", "4", "6", "8", "10"};

// Example: Sum up first ten squares of integers.
using namespace ranges;
int sum = accumulate(view::ints(1)
                    | view::transform([](int i){return i*i;})
                    | view::take(10), 0);
```

Undoubtedly, ranges are an enhancement worth waiting for. Yet, they are not going to completely replace iterators. Iterators will still be used to represent, for example, a single point in a sequence.

4. Filesystem

C++ allows the programmer to interact with multiple system resources, like CPU, RAM, or I/O devices. However, there is one resource that is not covered by the language – file system.

Nowadays it is hard to imagine an application that does not use a file system at all. It was observed that this kind of library would make a meaningful and desired functionality if it was available in C++. The aim of paper N4100 [3] is to introduce a file system library/support to the C++ language. Unsurprisingly, it is based on Boost.Filesystem. File systems are old and well known so the library itself does not contain any pieces that would be surprising. The library exports functions, classes, and enumerations that allow the programmer to effectively utilize file system resources like files, directories and so on. Exported classes and enumerations are briefly described in **Tables 1** and **2**, respectively. The rest of exported interface consists of free functions used to perform concrete actions like directory creation or file permissions inquiry.

Table 1 Classes provided by the filesystem library.

Class name	Purpose
path	Represents a path to file system entity. Contains pathname which can be invalid for external storage or operating system. Takes care of separators. Makes it easy to perform modifications like filename/file extension change and provides iterators for path traversal.
filesystem_error	Symbolizes an error that occurred during file system operation. Among other information it contains error message and optional paths related to the error.
file_status	Represents type and permissions of a file.
directory_entry	Stores a path object, which belongs to some directory.
directory_iterator	Input iterator allowing to iterate over entities stored in a directory.
recursive_directory_iterator	Same as directory_iterator, but iteration takes into account subdirectories.

Table 2 Enumerations provided by the filesystem library.

Enumeration	Description
file_type	Represents type of a file. Value equal to one of: none, not_found, regular, directory, symlink, block, character, fifo, socket, unknown.
copy_options	Enumeration used to specify bitmask which controls the copy process. These options are utilized in multiple situations like existence of target file or occurrence of symbolic links.
perms	Contains values used to create bitmask representing file permissions, like in POSIX-compilant operating system.
directory_options	Defines options used to control directory traversal.

Example usage of new filesystem library might look as presented below.

Listing 4 Example use of filesystem library.

```
#include <iostream>
#include <unordered_set>
#include <filesystem>

using namespace std;
using namespace filesystem;

int main()
{
    unordered_set<string> excludes = {
        "doc", "docs", "html"
    };

    recursive_directory_iterator dir(path(".")), end;
    while (dir != end)
    {
        auto path = dir->path();
        if (excludes.count(path.filename().string()))
        {
            dir.no_push(); // don't go inside
        }

        cout << dir->path().string() << endl;
        ++dir;
    }
}
```

It is clear that the new file system library, which is going to be a part of the standard C++ library, will enable programmers to cope with file system entities in an effective fashion. Based on the proposal paper one can say that the library is complete in terms of functionality. There is no doubt this is another big thing that a lot of developers are waiting for. It will positively change the C++ landscape.

5. Network

A few decades ago, when C++ emerged, most applications were designed to run on a single machine, utilizing its resources like one CPU core, some RAM memory, and a hard disk. A lot of things have radically changed ever since. Today, many applications are network-based and operate on virtualized hardware in the so-called clouds.

Writing this kind of applications requires programmers to cope with a network in some way. In popular languages other than C++ it was pretty easy because their standard libraries were shipped with network frameworks. With C++ it is different and programmers are forced to seek for a framework that would match their needs, be portable across operating systems etc. One of such frameworks is Boost.Asio (Asynchronous I/O). It is very likely to be a part of the C++ library from C++17 onwards, according to current status of proposal N4478 [2]. That would make C++ a network-aware language which is what many C++ programmers are waiting for.

Differences between networking library for C++17 and Boost.Asio are negligible, so it is possible to see what coping with a network would look like simply by using Boost.Asio. **Listing 5** illustrates a simple "Hello, world!" HTTP application.

Listing 5 Example HTTP "Hello, world!" application.

```
#include <iostream>
#include <asio/ts/internet.hpp>

using asio::ip::tcp;
using namespace std;

int main()
{
    asio::ip::tcp::iostream s;
    auto host = "example.com";

    s.connect(host, "http");
    if (!s)
    {
        cout << "Unable to connect: " << s.error().message()
        << endl;
        return 1;
    }

    s << "GET / HTTP/1.0"    << endl
      << "Host: " << host    << endl
      << "Accept: */*"       << endl
      << "Connection: close" << endl << endl;

    string response;
    getline(s, response);

    cout << "Response: " << response << endl;
}
```

Because the library is integrated with C++ streams it is extremely easy to send and receive data over the network. However, in the example shown above it happens in a synchronous fashion. Usually, this is not the approach used by applications that are meant to be blazingly fast because each network operation blocks the running thread. This, in turn, results in poor performance. Fortunately, the library provides asynchronous methods as well (**Listing 6**). The only visible drawback of this method is decreased readability.

Listing 6 Example UDP echo server.

```
#include <iostream>
#include <asio/ts/buffer.hpp>
#include <asio/ts/internet.hpp>

using asio::ip::udp;
using namespace std;

class server
{
public:
    server(asio::io_context & io_context, short port)
        : socket_(io_context, udp::endpoint(udp::v4(), port))
    {
        do_receive();
    }
private:
    void do_receive()
    {
        socket_.async_receive_from(
                asio::buffer(data_, max_length), sender_endpoint_,
                [this](error_code ec, size_t bytes_recvd)
                {
                    if (!ec && bytes_recvd > 0)
                    {
                        do_send(bytes_recvd);
                    }
                    else
                    {
                        do_receive();
                    }
                });
    }

    void do_send(size_t length)
    {
        socket_.async_send_to(
                asio::buffer(data_, length), sender_endpoint_,
                [this](error_code, size_t)
                {
                    do_receive();
                });
    }
private:
    udp::socket socket_;
    udp::endpoint sender_endpoint_;
    static constexpr short max_length = 1024;
    char data_[max_length];
};

int main()
{
    auto constexpr port = 4321;

    asio::io_context io_context;
    server s(io_context, port);
    io_context.run();
}
```

Implementation behind asynchronous interface utilizes well-known proactor design pattern, which has been named "executor" in the proposal.

6. Modules

The C++ language inherited lots of concepts and design ideas from its predecessor – the eternal C. One of the most fundamental and meaningful one is the system of independent compilation which, in essence, was about merging of multiple isolated translation units into one object file. Such object files are then subject of further processing. Translation units usually consist of multiple header files and one source file. This kind of design has some benefits like effectiveness, for instance, but suffers in other dimensions, like the following:

- Consistency between translation units is hard to maintain both for the programmer and the compiler (e.g. it is relatively easy to break One Definition Rule (ODR) without even knowing about it).
- The compiler has to do the dirty job of parsing header file content over and over again each time it is included in some translation unit and that results in a performance loss.
- IDEs have trouble understanding the code and helping the programmer, because as small things as preprocessor directives can radically change a lot of things.

The programming community has developed several techniques of mitigating some of those problems, but the solutions usually affect compilation or run-time performance (e.g pimpl idiom). Fortunately, the need for a clever approach has recently materialized in the form of standard proposal N4214 [1].

The proposal indicates very clearly that it would be too overwhelming to completely remove the preprocessor from C++ ecosystem. Therefore, the modules will live in one room with the preprocessor for some time. It was also observed that in C++ there are already seven scoping abstractions, so the authors have decided that module statement would not introduce a new scope.

In essence, the module system is about one unit exporting things inside a module that can be then imported by some other unit (possibly from another module). That is illustrated in **Listing 7**.

Listing 7 Example implementations utilizing module system.

```
// M1_interface.cpp
module M1;

export {
    class Foo { /*...*/ };
    class Bar { /*...*/ };
    // ...
}

// M1_impl_foo.cpp
module M1;

int Foo::foo() {
    // impl.
}

// M1_impl_bar.cpp
module M1;

int Bar::bar() {
    // impl.
}

// M2_interface.cpp
module M2;

export {
    class Baz { /*...*/ };
    // ...
}
```

The module system will provide easy componentization and separation. It will coexist with header files but will allow mitigating of their usage in C++ applications. There are three keywords to be associated with the module system:

- Module – used to indicate to which module a given translation unit belongs
- Export – used to specify declarations that should be exported from the module
- Import – used to import the entity exported within another module

Importing of objects exported by other modules is illustrated in **Listing 8**.

Listing 8 Example of importing objects exported by the modules. [1]

```
import std.vector;   // #include <vector>
import std.string;   // #include <string>
import std.iostream; // #include <iostream>
import std.iterator; // #include <iterator>
int main() {
  using namespace std;
  vector<string> v = {
    "Socrates", "Plato", "Descartes", "Kant", "Bacon"
  };
  copy(begin(v), end(v), ostream_iterator<string>(cout, "\n"));
}
```

Those changes may seem simple and meaningless, but they will in fact enable programmers to work with huge C++ projects in a more effective fashion, the main reason being higher compilation speeds and tailored build frameworks. If the module system is included in C++17, it will surely be a real game changer.

7. Summary

More and more things are continuously becoming part of the C++ language, leaving less space for operating systems and third party libraries. Unless there is a library tailored for developing concrete solutions, this looks like a good direction.

C++ is definitely catching up after its competitors. With brand new libraries like the ones presented in the article programmers will be able to quickly write applications that utilize all the resources that a modern application has to utilize (like network). Some other libraries and language features will make life easier and allow writing a code in a more functional way.

There is no doubt that C++17 will change the way people perceive C++ as a language. This renewed language will definitely be more attractive than it is now.

References

[1] http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4214.pdf
[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4478.html
[3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4100.pdf
[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4382.pdf
[5] https://ericniebler.github.io/range-v3/

About the author

I am a Software Engineer with years of experience.
I started my adventure with computers at the age of 12 with Commodore64 as a friend. Several years ago I joined the community that try to master C++ language.
My leisure time is almost indistinguishable from office hours. I am interested in novelties in computer science, algorithms, distributed systems, programming languages, etc.

Sławomir Zborowski
Engineer, Software Development C++
MBB Single RAN

Professional Software Development

Make It Simple: Java Generics

Krzysztof Bulwiński
Software Engineer
MBB CEM & OSS

NOKIA

Introduction

A generics mechanism was introduced with Release 5 of Java Standard Edition (JSE), and since then it has become one of the key components in the language syntax. It is probably safe to say that every Java programmer must have encountered it at some point along his coding path. A good example here may be a parameterized collection.

Despite the fact that there is a lot of information out there, some developers do not understand the meaning and implications of Java generics. This paper discusses the concept of Java generics in the simplest possible way; it is an attempt to create an easy to understand guide for developers who might find generics somewhat confusing.

In software projects, a bug occurrence is an obvious fact. It is true that thorough programming and testing reduces the probability of faults, but the golden principle of testing states that there is no way to test everything. Therefore, bugs will somehow always find their way to crawl into the production code. This becomes more and more evident as the project grows.

Compile time bugs can be detected at a very early stage. The compiler comes in very handy here. Runtime bugs, on the other hand, are much more problematic as they are not so easy to notice.

Generics introduces stability to the development code by eliminating some of the bugs right at the compile time. To simplify matters, all imports and almost all access modifiers in the code snippets are omitted as they are not required to understand the examples. On the other hand, they may unintentionally obscure the whole picture. The code snippets presented throughout this paper are based on Java SE 8.

1. Motivation behind the generics

As mentioned above, the generics mechanism was unknown before Release 5 of Java. Thus, it may bring up questions about what problems it caused and how developers managed to overcome them. To take a closer look at the issue, let us consider two distinct objects in Java terms, a dog and a car, for instance. In the Java world, they can be modeled by means of two classes as follows:

Listing 1.1

```
class Dog
{
  void makeSound()
  {
    System.out.println("Dog barks.");
  }
}
```

```
class Car
{
  void drive()
  {
    System.out.println("Car drives.");
  }
}
```

Now, let those objects be added to the collection as follows:

Listing 1.2

```
void putDogCarToCollection(Collection items)
{
  items.add(new Dog());
  items.add(new Car());
}
```

It may be noticed that the collection can be filled with a variety of objects that have nothing in common from an implementation point of view. Now, let us attempt to read the objects from the collection as follows:

Listing 1.3

```
void fetchDogCarFromCollectionDoAction(Collection items)
{
  //Collections items treated as java.lang.Object.
  for (Object item : items)
  {
    //Type checking required
    //to cast objects safely.
    if (item instanceof Dog)
    {
      Dog dog = (Dog) item;
      dog.makeSound();
    }
    else if (item instanceof Car)
    {
      Car car = (Car) item;
      car.drive();
    }
  }
}
```

The code to be tested as a whole is as follows:

Listing 1.4

```
void runIt()
{
  Collection items = new ArrayList();

  putDogCarToCollection(items);
  fetchDogCarFromCollectionDoAction(items);
}
```

What is the actual issue in the example presented above? In **Listing 1.2**, everything appears to be unproblematic; it is a simple operation of adding a variety of objects to the collection, nothing out of the ordinary. The issue reveals itself in **Listing 1.3**, when implementing a *for-each* loop. All objects fetched from the collection are treated as *java.lang.Object*. Therefore, in order to be able to invoke methods of a specific type, they must be transformed beforehand to become specific objects again. In reality, it is done by telling the compiler that the object we are dealing with is of a specific type, which is achieved by a casting operation. However, unless it is performed carefully, Java may throw a *java.lang.ClassCastException* exception. Therefore, to perform the operation safely, an *instanceof* operator must be utilized together with an *if* clause. When dealing with a small number of types, this is not much trouble, but adding a great number and variety of object types introduces unnecessary chaos in the code (*instanceof's* and *if's*) as well as in the logical understanding of such a structure. Furthermore, it becomes hard to maintain and debug. Typically, collections are intended to store items of one type for the obvious reason of making the code clean and logically cohesive.

To take the burden of keeping the type right off the developer, there comes the generics feature. It offers the confidence that the object added and fetched to and from the collection is of a specific type. It is the compiler that keeps track of the type that is to be put into the collection. How it is done in practice will be explained in greater detail in the remainder of this paper.

2. Generics as a type variable

Generics introduces the concept of a type variable, which, in other words, is an unqualified identifier introduced by:

- generic class and interface
- generic methods and constructors

2.1. Generic class and interface declaration

A class or an interface is a generic type if it is parameterized over types; in other words, if it has one or more type variable(s). In the language syntax, a variable type is delimited by angle (<>) brackets following the class or interface name.

Generally speaking, type variables play the role of parameters and deliver information to the compiler that there is a need to perform type checking. An example of a generic class declaration begins with a class name, followed by a capital letter T closed in angle brackets. Generic methods will be explained in greater detail below.

Listing 2.1

```
class GenericClass<T>
{
  private T item;

  //Getter and setter methods omitted.
  public static void main(String[] args)
  {
    GenericClass<Dog> dogGeneric = new GenericClass<>();
    dogGeneric.setItem(new Dog());
    Dog dog = dogGeneric.getItem();
    dog.makeSound();
  }
}
```

It may be noticed that the main method has no type checking, i.e. no *instanceof* operator used. Additionally, there is no object casting; therefore, the fear of a class cast exception may be mitigated here. An example of a generic interface declaration begins with an interface name, followed by a capital letter T closed in angle brackets.

Listing 2.2

```
public interface GenericInterface<T>
{
  T getItem();
  void setItem(T item);
}
```

Many items in the Java Application Programming Interface (API), such as the entire collections framework, are adjusted to utilize generics extensively. A good example to confirm that is the *java.lang.Comparable* interface. As a result, code snippets presented in section 1 can easily be modified to use generics.

Adding items to the collection:

Listing 2.3

```
void putDogToCollection(Collection<Dog> items)
{
  //Only dogs allowed.
  items.add(new Dog());
}

void putCarToCollection(Collection<Car> items)
{
  //Only cars allowed.
  items.add(new Car());
}
```

Fetching items from the collection:

Listing 2.4

```
void fetchDogFromCollectionDoAction(Collection<Dog> items)
{
  for (Dog dog : items)
  {
      //No type checking required.
      dog.makeSound();
  }
}

void fetchCarFromCollectionDoAction(Collection<Car> items)
{
  for (Car car : items)
  {
      //No type checking required.
      car.drive();
  }
}
```

The code to be run as a whole is as follows:

Listing 2.5

```
void runIt()
{
  Collection<Dog> dogItems = new ArrayList<>();
  putDogToCollection(dogItems);
  fetchDogFromCollectionDoAction(dogItems);

  Collection<Car> carItems = new ArrayList<>();
  putCarToCollection(carItems);
  fetchCarFromCollectionDoAction(carItems);
}
```

In the current situation, the code will simply not compile when there is an attempt to put a car object into the collection of dogs and vice versa. It may be noticed that adding and fetching objects to and from the collection introduced a logical separation based on a particular object type. This, as a matter of fact, is a positive occurrence. It is worth noting how significantly the source code has been simplified; now, essentially each method deals only with one type of object. What is more, neither casting nor an explicit type checking is required.

2.2. Generic methods and constructors declaration

This section provides a detailed description of how to utilize generics with respect to methods and constructors.

2.2.1. Generic methods

A given method becomes generic if it declares one or more type variables:

Listing 2.6

```
public <T> T getFirstItem(List<T> items)
{
  //Method parameter treated normally.
  if (items == null || items.isEmpty())
  {
    return null;
  }
  else
  {
    //Fetching first element of the collection.
    //Type is not interesting at the moment.
    return items.get(0);
  }
}
```

In the trivial code snippet in **Listing 2.6** above, the method accepts a collection of items and returns the first element of the collection. Interestingly, at the compile time, the type is unknown, and any type of object fits. It is worth noticing that the method parameter requires no special treatment while:

- *null* reference checking
- verifying whether the collection is empty
- returning the value of the first collection element

The fact that they are generic is not significant in this particular case.

Listing 2.7

```
void runIt()
{
  List<Dog> dogs = new ArrayList<>();
  dogs.add(new Dog());
  //Generic method used for dogs.
  Dog firsDogItem = getFirstItem(dogs);
  firsDogItem.makeSound();

  List<Car> cars = new ArrayList<>();
  cars.add(new Car());
  //Generic method used for cars.
  Car firstCarItem = getFirstItem(cars);
  firstCarItem.drive();
}
```

Listing 2.7 explains why generics is so powerful. A method taking an argument of any type and solving it during the runtime may be reused in the application's code. It leads to a significant reduction of code multiplications, which in turn yields in a much cleaner and reusable code. This result is a highly desirable one, especially in big scale applications.

2.2.2. Generic constructors

A constructor becomes generic if it declares one or more type variable(s). In **Listing 2.1.**, it may be noticed that the instance variable *item* was set by means of the setter method. However, the member can also be set in the constructor at an object's initiation stage.

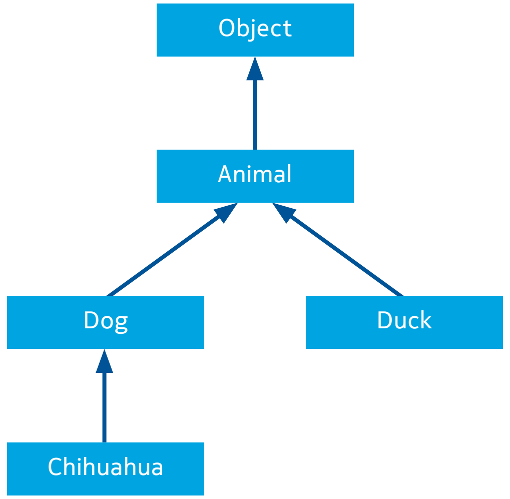
3. Subtyping issue and generic wildcards

This section explains the difference between a Java object type and generics subtyping. Those two terms are very often misunderstood.

3.1. Subtyping generics: is it really an issue?

Java developers are aware of the fact that it is safe to assign an object of one type to an object of another type as long as the types are compatible. To be more precise, **Figure 1** depicts an inheritance structure which represents the relation between Java objects.

Figure 1 Example of simple inheritance.



Listing 2.8

```
class GenericClass<T>
{
  private T item;

  //Generic instance variable
  //set in the constructor.
  GenericClass(T item)
  {
    this.item = item;
  }

  //Getter code omitted.

  public static void main(String[] args)
  {
    GenericClass<Dog> dogGeneric =
                           new GenericClass<>(new Dog());
    Dog dog = dogGeneric.getItem();
    dog.makeSound();
  }
}
```

In the example:

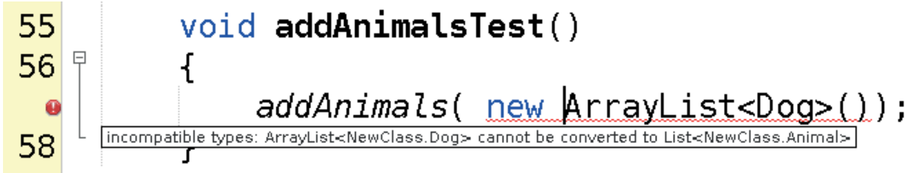
- Chihuahua is a subtype of a Dog.
- Dog is a subtype of an Animal.
- Chihuahua is a subtype of an Animal.
- Duck is a subtype of an Animal.
- They are all a subtype of an Object.
- Nevertheless, a Duck is not a subtype of a Dog and Chihuahua.

In the object-oriented terminology, the relation is called "*is-a*" relation. In other words, a Chihuahua is-a Dog, a Dog is-a Animal, etc. Therefore, in the Java code, it is safe to perform the following assignments:

Listing 3.1

```
class Animal {}

class Dog extends Animal {}

class Chihuahua extends Dog {}

class Duck extends Animal {}

void isaRelationship()
{
  //Is-a relationship examples.
  Object object = new Object();
  Animal animal = new Animal();
  Dog dog = new Dog();
  Chihuahua chihuahua = new Chihuahua();
  Duck duck = new Duck();

  animal = dog;
  animal = chihuahua;
  animal = duck;
  dog = chihuahua;

  object = animal;
  object = dog;
  object = chihuahua;
  object = duck;
}
```

Figure 2 Compilation error: incompatible types.



```
void addAnimalsTest()
{
    addAnimals( new ArrayList<Dog>());
incompatible types: ArrayList<NewClass.Dog> cannot be converted to List<NewClass.Animal>
}
```

As for generics, the same principles apply here. The code snippet in **Listing 3.2** explains the principle:

Listing 3.2

```
List<Animal> animals = new ArrayList<>();
animals.add(new Animal());
animals.add(new Dog());
animals.add(new Chihuahua());
animals.add(new Duck());
```

Adding items to the collection in such a manner is safe. After all, as they are all Animal, the collection will accept all of its subtypes without any reservations. Obviously, fetched objects from the list will be a type of Animal. Hence, having a specific object, casting operation, and type checking is required here. The example appears simple and understandable. The situation, however, looks completely different when dealing with such an example as in **Listing 3.3**.

Listing 3.3

```
void addAnimals(List<Animal> animals)
{
  //Implementation omitted
}
```

What kind of arguments are acceptable? The answer may seem obvious at first, a list of animals. Does this mean that it is possible to pass a list of Dogs, Chihuahuas, or Ducks? Such an answer seems obvious based on what was expected from previous explanations. Unfortunately, the answer is no, because *List<Dog>* and *List<Chihuahua>*, etc. are not subtypes of *List<Animal>*.

In order to clarify, it is important to differentiate two things. First, attempting to pass, e.g. a list of Dogs to the method from **Listing 3.3** results in a compilation error as presented in **Figure 2** below.

Second, adding Dogs, for example, to the list of Animals is acceptable as the following code snippet depicts (*addAnimals* method from **Listing 3.3**).

The code to be tested as a whole is as follows:

Listing 3.4

```
void addAnimalsTest()
{
  //Creating list of animals.
  List<Animal> animals = new ArrayList<>();
  //Adding a dog to the list.
  animals.add(new Dog());
  //Passing the animals list to the method.
  addAnimals(animals);
}
```

This is often misunderstood by developers when it comes to writing code that uses generics. The problem can easily be solved by means of wildcards. In the Java language syntax, the operator is represented by means of a question mark (?) and means an unknown type. It can be utilized in a variety of situations; this paper, however, focuses specifically on the following aspects: unbounded, lower and upper bounded wildcards.

3.2. Upper bounded wildcards

An upper bounded wildcard is used to relax the restriction on a variable. To declare it, a wildcard character (?) should be used, followed by the *extends* keyword, followed by the base type. To make the method from **Listing 3.3** work with a list of Dogs, Ducks, and Chihuahuas, the following modification shown in **Listing 3.5** is required:

Listing 3.5

```
void addAnimals(List<? extends Animal> animals)
{
  //Implementation omitted.
}
```

It is now safe to feed the method with the list of types that inherit from the Animal. Objects taken out from such a collection are cast to the upper type from the inheritance hierarchy; in the presented case, it is the Animal type.

Listing 3.6

```
void addAnimalsTest()
{
  //Creating list of Dogs.
  List<Dog> dogs = new ArrayList<>();
  //Adding a Dog to the collection.
  dogs.add(new Dog());
  //Method accepts wildcard type.
  addAnimals(dogs);
}

void addAnimals(List<? extends Animal> animals)
{
  for (Animal animal : animals)
  {
    //Objects in collection cast to upper type.
  }
}
```

Upper bounding simply means imposing the upper type restriction of the types to be accepted.

3.3. Unbounded wildcards

The unbounded wildcard type is declared using a wildcard character (?); for example, *List<?>* means a list of an unknown type. The unbounded wildcard approach is useful when:

- implementing a method which uses API of the *java.lang.Object* class
- The code does not depend on the type. A good example here is the *java.util.List.size* method; there is no need to know the type of the collection in order to count objects.

Listing 3.7

```
void unboundedWildcard(List<?> list)
{
  for (Object listItem : list)
  {
    //Implementation omitted.
  }
}
```

The objects taken out of the unbounded type collection can be cast to the specific ones. Of course, an instance type checking is strongly advised beforehand. An unbounded wildcard, in other words, means no type restrictions.

3.4. Lower bounded wildcards

The upper bounded wildcard restricts the unknown type to be a specific type of a subtype of that type and is specified by the *extends* keyword. In a similar manner, a lower bounded wildcard restricts the unknown type to be a specific type or a *super* type of that type. To declare it, a wildcard character should be used (?), followed by the *super* keyword, followed by its lower bound type. Using the inheritance tree presented in **Figure 1**, let us assume that there may be a need to implement a method that accepts a list of *java.lang.Object* Animals and Dogs. This can be achieved by means of the lower bounded wildcard type as follows:

Listing 3.8

```
void lowerBounded(List<? super Dog> objects)
{
  for (Object object : objects)
  {
    //Implementation omitted.
  }
}
```

Objects stored in such a bounded collection are cast to the *java.lang.Object* type. It may seem unusual at first that the collection of a specified type returns objects as *java.lang.Object* objects. However, on second thoughts, it seems to be a reasonable approach. Such a collection can accept anything that is upward compatible with the Dog type. Therefore, in order to avoid ambiguity, it is safe to cast any object to the *java.lang.Object* type. After all, the compiler does not know what specific type to expect. Naturally, collection items can be easily converted to its specific type. The method from **Listing 3.8** can be invoked as follows:

Listing 3.9

```
//Adding list of Dogs.
lowerBounded(new ArrayList<Dog>());
//Adding list of Animals.
lowerBounded(new ArrayList<Animal>());
//Adding list of Objects.
lowerBounded(new ArrayList<Object>());
```

Lower bounding simply means imposing the lowest possible type in the inheritance hierarchy restriction of the types to be accepted.

Conclusion

The purpose of this paper was to make generics easier to understand for those who may find it confusing. Discussed issues were supported by straightforward examples in order to present the subject matter as easily comprehensible as possible. It is my hope that after reading this paper and analyzing the examples, generics will start being utilized with more awareness, especially by those who have often used it after a cursory glance at some tutorials without actually understanding it. Generics can be a very powerful tool, and as with every tool, whenever it finds itself in skillful hands, good things may come out of using it.

References

[1] https://docs.oracle.com/javase/tutorial/java/generics/types.html

About the author

I am a graduate of the Electronics faculty of Wrocław University of Technology. I work as a Java Software Engineer in the MBB CEM & OSS department. I am involved in developing software that offers end-to-end solutions in the areas of configuration, and fault and performance management for mobile networks operators. The job is very interesting even though challenging at times. All in all, it is good to feel that the work we do in our department actually serves a good cause.

Krzysztof Bulwiński
Software Engineer
MBB CEM & OSS

Functional Reactive Programming Paradigm in JavaScript

Bartosz Kwaśniewski
Software Developer
MBB Single RAN

Introduction

The purpose of this paper is to describe the Functional Reactive Programming (FRP) paradigm and verify how well the language of JavaScript (JS) supports it. The existing literature on FRP, its implementations, benefits, and uses, has been reviewed, and an overview of JavaScript paradigms, concepts, and properties has been provided to precisely state what a paradigm is and how JavaScript supports it.

FRP originated from an animation library FRAN, hosted in a purely functional and strong-typed Haskell language. JavaScript, contrary to Haskell, is a multi-paradigm language, non-pure, without type checking. Thus, a question arises if JavaScript might be used to program reactive systems in a functional way. The advantages of FRP, JavaScript facilities, and its existing libraries would need to be verified.

JavaScript is "The World's Most Misunderstood Programming Language" [28] because of its: misleading name, Lisp in C's clothing, typecasting, design errors, lousy implementations, bad books, substandard standard, amateur programmers, and fake object-orientation. To complete Douglas Crockford's list, the author of *JavaScript: Good Parts* [29], it seems appropriate to mention the existing variety of programming styles that a programmer can freely intermix.

A good language for large programs should support several paradigms because different issues require different concepts to solve them. According to MDN [38], "JavaScript is a lightweight, interpreted, object-oriented language with first-class functions and closures. It is known as the scripting language for Web pages, but also used in many non-browser environments as well such as Node.js." *Functional JavaScript* [12] defines JS as "a prototype-based multi-paradigm scripting language that is dynamic, and supports object-oriented, imperative, and functional programming styles."

1. Programming languages, paradigms, and concepts

Programs can be large, reaching millions of lines of source code, written by large teams over many years. In order to establish the level of support of a given language for a specific paradigm, paradigms and programming concepts need to be described, and certain underlying conditions of how to construct such systems need to be determined.

Each issue has a paradigm that is best for it, and that is why it is important to choose carefully the paradigms supported by the language. Programming paradigms are built out of programming concepts [20]:

- Lexically scoped closures. It is a powerful concept which the central paradigm, a functional programming, is founded upon. Many abilities normally associated with specific paradigms are based on closures:
  - Instantiation and genericity, associated with object-oriented programming, can be done by writing functions that return other functions.
  - Separation of concerns, associated with aspect-oriented programming, can be done by writing functions that take other functions as arguments.
  - Independence. Constructing a program as independent parts. When two parts do not interact at all, we say they are concurrent.
- Named state. State introduces an abstract notion of time in programs. It is important for a system's modularity, a concept when updates can be done to a part of the system without changing the rest of the system.
- Data abstraction and interfaces. It is a way of organizing the use of data structures according to precise rules which guarantee that the data structures are used correctly.

There are different levels of support for paradigms by different languages [22]:

- A language supports a style of programming if it provides facilities that make it convenient, reasonably easy, safe, and efficient to use.
- A language merely enables the technique to be used.
- A language does not support a technique if it takes exceptional effort or skill to write such programs.

Support for a paradigm comes not only in the obvious form of language facilities that allow a direct use of paradigms, but also in the form of:

- extended linguistic support for paradigms that are against unintentional deviation from the paradigm
- extra linguistic facilities such as libraries and programming environments

The important issue is not how many features a language possesses, but that the features it does possess are sufficient to support the desired programming styles. It is not enough that libraries have been written in the language to support the paradigm. The language itself should support the paradigm. FRP is an intersection of two paradigms: reactive programming and functional programming.
The reactive programming style is oriented around data flows, concurrency, the propagation of change, and event-based and asynchronous systems. Reactive systems are systems that continuously react to stimuli coming from the environment by sending back responses. Imperative techniques to create reactive systems, such as the observer pattern, lead to a plethora of problems: inversion of control, non-modularity, and side effects [6].

2. Functional programming

Functional programming is a declarative programming paradigm in which computation is carried out entirely through the evaluation of expressions, characterized as having no implicit state. It is often described as expressing what is being computed rather than how, in contrast to imperative languages, which have an implicit state that is modified by a sequence of commands.

Functional languages originated from the lambda calculus, called the smallest universal programming language of the world. Then comes Lisp, Iswim, ML, and finally in 1987 [1] Haskell. The purpose of Haskell was to consolidate the existing functional languages into a common one that would serve as a basis for future research and would popularize the functional paradigm.

The procedure for a functional language is simple: just drop the assignment statement; however, it is better to characterize a language by pointing out the features it does have instead of those it is lacking in [23]:

- Higher-order functions. Functions that are like any other values, and additionally they could be passed as arguments or returned as results. It is the primary abstraction mechanism over values. It increases modularity by serving as a mechanism for "glueing" program fragments together [26].
- Non-strict semantics and lazy evaluation (call-by-need). A key feature is that arguments in function calls are evaluated at most once, which frees a programmer from concerns about an evaluation order. The primary power of lazily evaluated data structures comes from their utility in separating data from control.
- Equations and pattern-matching. Using equations as part of the syntax is a way of making functional programming look concise and elegant. Equations could be considered as definitions of functions. It is a way of calling a different function based on a different context. There could be several equations defining the same function, only one of which is presumably applicable in a given situation.
- Data abstraction mechanisms. Data abstraction is a way of organizing the use of data structures according to precise rules which guarantee that the data structures are used correctly. It improves modularity, security, and clarity of programs, which helps to write large programs:
  - Modularity is improved because one can abstract away from implementation details, and implementation could be divided into separate parts developed by different people.
  - Security is improved because the interface defines the authorized operations on the data structures, no other operations are possible, and its violations are automatically prohibited.
  - Clarity is improved because data abstraction has an almost self-documenting flavor and creates simplified models not obscured with details.
- Purity and referential transparency with equational reasoning. Purity, the lack of side effects, accounts for the primary ability to apply equational reasoning because a pure language possesses the property of referential transparency. If a function is called twice with the same parameters, it is guaranteed to return the same result both times.
- Strong static typing and type inference. Types create a formal semantics for a better understanding of programming languages and add additional constraints to it. Possible errors can be caught at the compile time. Type inference refers to an automatic deduction of a data type, which makes many programming tasks easier while still permitting type checking.

3. Functional reactive programming (FRP)

FRP can be understood as a network of functions that propagate incoming events. Each node is a function, a reaction to an event, and a switch that propagates an event or new events further. Programming such systems is about designing networks of functions. There are two ways of doing it: first, directly, when a host language supports the FRP paradigm natively, as it is in Haskell; second, indirectly, with the help of libraries in a language that lacks such facilities. In an interface, the library exposes methods called combinators, which add or modify the nodes by changing the routing of events (see **Figure 1**). The main property is that the FRP system must be deterministic, without side effects.

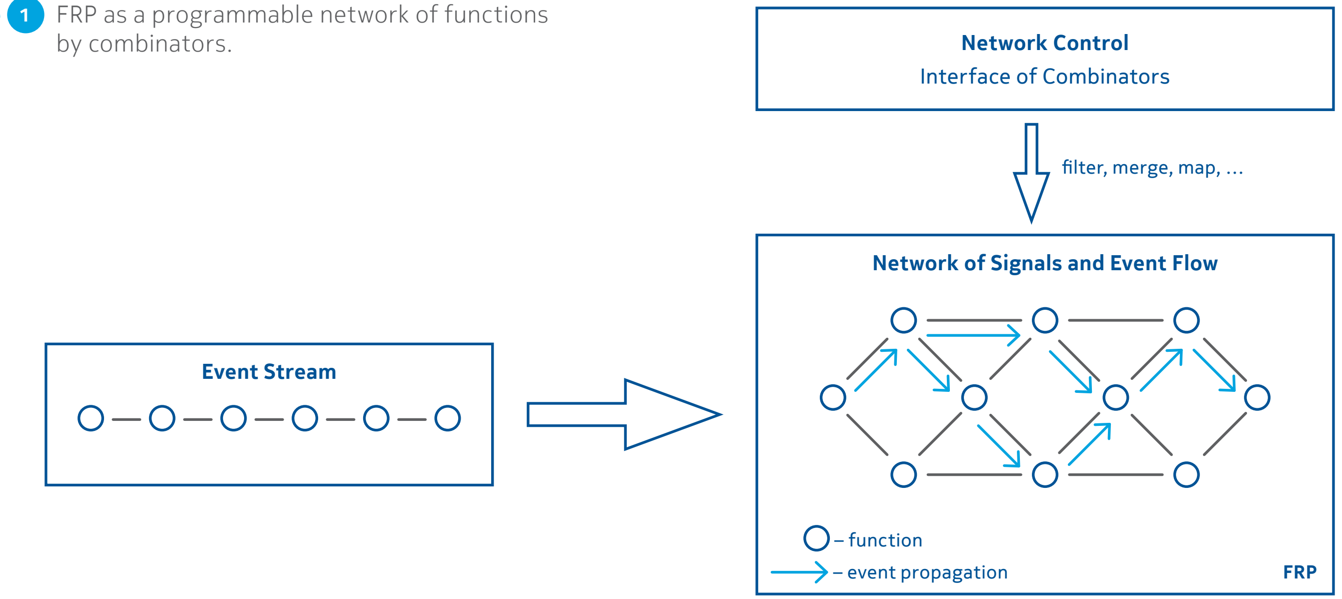
The understanding of FRP has evolved throughout time, from a DSL language to a programming style based on event streams. There are many views and definitions of what FRP is exactly: a data-flow paradigm [16], a conceptual framework [2], a method of modelling reactive behaviour [2], a useful model [27], an idea [24], a paradigm extending functional languages [3], a paradigm for programming hybrid systems [5], a declarative programming model [10], a library of functions and types that extend Haskell [18], a declarative domain-specific language [21], an approach to reactive programming [15], or a continuous synchronous programming [20].

FRP was originally developed in Functional Reactive Animation (Fran) [13], a Domain-specific Embedded Language (DSEL), embedded in Haskell. Without bias toward application specifics, it is being developed mainly by the Yale Haskell Group [35]. FRP was used as DSELs in many application domains: robotics (Frob), graphical user interfaces (Fruit), parallel programming (HPorter), networking (Nettle), and computer music (Euterpea). Other uses include general signal processing, GUI tool-kits, simulators, Web programming [4, 9, 17], and almost any embedded application.

It is hard to distinguish the core of FRP from the features of Haskell language itself. FRP was pushed towards real-time embedded systems by several variants, including Real-Time FRP [21] and event-driven FRP [8]. The core ideas of functional reactive programming culminated in Yampa [5, 36, 24], based on an original idea of Fran.

The goal of FRP implementations is to enable: safe programming (compile time checking), efficient programming (real-time), and

Figure 1 FRP as a programmable network of functions by combinators.



composability (a functional framework built of smaller modules piecewise). The origins of FRP inspired a wide variety of reactive systems and libraries in many areas and languages: C++ [11, 31], Python [19], Java [10], JavaScript [9, 32], and Scala [6], or F# [33].

3.1 FRP concepts and semantics

FRP integrates reactivity directly into the functional programming style while hiding the mechanism that controls the time flow under an abstraction layer. It supports both continuous and discrete time varying values: a signal or behavior is an abstraction of a continuous and time-varying value, an event is an abstraction of occurrence, and an event stream is an abstraction of discrete incoming and outcoming changes. Reactivity is achieved by providing constructs for specifying how signals change and propagate in response to events (event stream). A system is described as signal-processing networks; it is described in terms of functions mapping signals to signals. The synchronous data-flow principle and support for both continuous and discrete time are common for all variants of FRP. Three distinct semantic frameworks have emerged:

- Classic FRP. Signals (behaviors) and events are first-class values which are directly manipulated by various language constructs.
- Unary FRP (Signal functions). Its semantics includes the concept of signals but does not include them as first-class values or reactive constructs. More exactly, functions on signals are manipulated and made reactive. Events are represented as a special case of signals and manipulated with specialized signal functions.
- N-ary FRP. It is an improvement on signal functions, where signal functions are not functions from signal to signal but rather signal vector to signal vector.

3.2 FRP example in Yampa

Yampa (Arrowized FRP) is an instantiation of FRP as a domain-specific language embedded in Haskell. Its most characteristic feature is that the core FRP concepts are represented using arrows, a generalization of monads. The programming discipline induced by arrows prevents certain kinds of time-leaks and space-leaks that are common in generic FRP programs, thus making Yampa more suitable for systems having real-time constraints (see **Figure 2**).

The basic concept is a signal function, a mapping from an input signal to an output signal. The Signal Function (SF) type (see **Listing 1**) is made an instance of the Arrow class. The Yampa program expresses the composition of a possibly large number of signal functions into a composite signal function.

Listing 1 Signal function.

```
-- Signal Function
SF a b = Signal a -> Signal b
```

A minimal universal set of combinators is sufficient to express all possible wirings. The first three combinators constitute a minimal universal set (see **Listing 2**).

Listing 2 Primitive operators and combinators.

```
-- primitive operator lifting normal function to Signal Function
arr :: (a -> b) -> SF a b
(>>>) :: SF a b -> SF b c -> SF a c

-- primitive combinator to compose signal functions
(&&&) :: SF a b -> SF a c -> SF a ( b, c)
```

A successful demonstration of the performance of signal function implementations was an implementation of the classic game "Space Invaders" in Yampa. Using this framework, together with looping combinators, the game was implemented in a manner of objecto-riented programming, where each game object was represented by a signal function, and objects could pass messages to each other as well as respond to external input.

A simple physical model and a control system for a gun can be specified in just a few lines of Yampa code (see **Listing 3**):

Listing 3 Example of Yampa usage.

```
data SimpleGunState = SimpleGunState {
  sgsPos :: Position2,
  sgsVel :: Velocity2,
  sgsFired :: Event ()
}

type SimpleGun = SF GameInput SimpleGunState

simpleGun :: Position2 -> SimpleGun
simpleGun (Point2 x0 y0) = proc gi -> do
  (Point2 xd _) <- ptrPos -< gi
rec

-- Controller
let ad = 10 * (xd – x) – 5 * v

-- Physics
v <- integral -< clampAcc v ad
x <- (x0 +) ^<< integral -< v
fire <- leftButtonPress -< gi
returnA -< SimpleGunState {
  sgsPos = (Point2 x y0),
  sgsVel = (vector2 v 0),
  sgsFired = fire
}
```

4. JavaScript: overview

JavaScript is considered a problematic language for developing large scale applications; it is not yet clear what should be done about it [25]. It is essential to understand what exactly these shortcomings are and why they should be remedied.

4.1 JavaScript: shortcomings and remedies

- Bad scoping semantics. All variables live in the global scope, unless explicitly declared local. Curly braces do not open a new scope. The only local scope available is at the function level.
- Weak typing. While a static type system is often a great help in finding errors and reasoning about code, the real problem is that if an operation is applied to values with mismatching types, rather than throwing an error, the runtime silently attempts to convert one or more of the values into something which is compatible with the other. This has interesting consequences, e.g. the equality operator not being transitive.
- Poor support for the functional paradigm. If you define a functional language as a language that supports closures, first-class and higher-order functions, then yes, JavaScript is a functional language. However, if you also consider such factors as support for immutability, purity, algebraic data types, strong types, partial application, lazy evaluation, pattern matching, and equitational look, then no, JavaScript is not a functional language. JavaScript merely enables the functional programming style to be used because it lacks facilities that make it convenient and safe to use that style, and there is no linguistic that guards unintentional deviation from the paradigm. Fortunately, there are extra-linguistic facilities such as libraries, which help developers to use the functional style.
- Lacking modularity. JavaScript does not support modules or include files, making it extremely cumbersome to link individual pieces of code together.

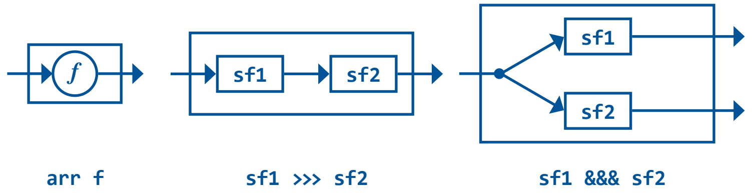
JavaScript is a flexible language, which makes extending it easy.

4.2 JavaScript as a Functional Reactive Language

JavaScript merely enables functional programming to be used. Because JavaScript lacks facilities that make it convenient, safe, and efficient to use that style, there is neither core nor an extensive linguistic support for FRP. However, there are extra linguistic facilities such as libraries, which enable JavaScript support for the FRP paradigm:

- Flapjax [9]. Flapjax is a programming language designed for client-based Web applications.
- RxJS. Reactive Extensions for JavaScript is a library for transforming, composing, and querying streams of data.
- BaconJS. Transforms data streams with a *map* and *filter*, combines with *merge* and *combine*. [37].

Figure 2 Common combinators.



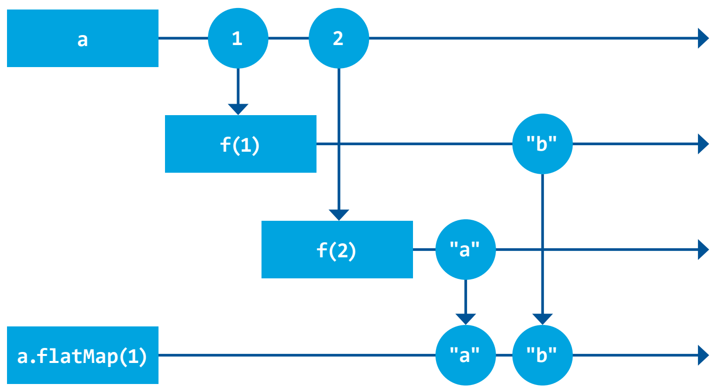
The most promising library is BaconJS. It makes use of combinators to design a network of event propagations. The fundamental abstractions here are observable event streams. Streams represent a series of multiple events. Using streams allows a simple expression of behavior that spans over time and multiple events. The FRP in the sense of JavaScript is understood as an event-stream processing; it is understood that streams are transformed into reactions to incoming events.

4.3 FRP example in BaconJS

Bacon.js is a reactive programming library that might be used on a client, server, or in a game programming. Everything becomes an asynchronous data stream: database on the server, mouse events, promises, and server requests. This lets you avoid what is known as "the callback hell" and enables you to handle errors better and compose streams together, which in turn gives you a better control and flexibility. This is achieved by various combinators and observers.

Combinators change the route of event propagation by filtering, merging, mapping, switching, etc.; they are language constructs which shall not break the purity. The common combinators in BaconJS are: *filter*, *map*, *scan*, *take*, *skip*, scan, *first*, *last*, *flatMap*, *fold*, *diff*, *reduce*, *zip*, *combine*, *concat*, and many more. Observers are the place where we apply side effects by changing a state, where we write an output. The observers are: *onValue*, *onError*, *onEnd*. However, JavaScript does not have any constructs to prevent the developer from violating those requirements. In **Listing 4**, there is an example of usage of *flatMap* (see **Figure 3**) and *filter* combinators processing the stream to deliver an event value after a timeout or not deliver it at all if there is a cancel with a matching id before the timeout.

Figure 3 Example of flatMap combinator.



Listing 4 Example of BaconJS usage with some combinators.

```
var incomingStream = Bacon . sequentially (100, [
  { id: 1, timeout : 300, type : "start" , value : 1},
  { id: 2, timeout : 300, type : "start" , value : 2},
  { id: 3, timeout : 400, type : "start" , value : 3},
  { id: 2, timeout : 300, type : "cancel" , value : 4},
  { id: 1, timeout : 900, type : "cancel" , value : 5},
  { id: 1, timeout : 300, type : "start" , value : 6}
]);

(function process ( stream ) {
  return stream
   .filter ( function (ev) {
     return ev. type === "start";
   })
  .flatMap ( function ( evStart ) {
     return Bacon
      .later ( evStart . timeout , evStart )
      .takeUntil ( stream
        .filter ( function ( evCancel ) {
           return evStart .id === evCancel
                  .id && evCancel.type == "cancel";
         })
        .take (1)
    );
  })
  .map ( function ( ev){
    return ev. value ;
  })
})(incomingStream)
 .onValue(function(val){console.log(val);}) //1,3,6
```

Conclusion

FRP is more than a paradigm; it is a technique or framework to write reactive systems in a more scientific way. It is promising, but not yet a mature technology.

There is an ongoing research in the FRP field. The first book about FRP, *Functional Reactive Programming* [14], is about to be published and FRP frameworks for the Web have recently emerged, e.g. Elm [34]. Unfortunately, there is still not much news about large systems successfully developed with FRP paradigms. FRP in JavaScript is also a new research topic, and only about Web and GUI development [4, 9, 7, 12, 17, and 25].

JavaScript's support for FRP is on a library level; it is not direct, as in the case of Haskell. Other languages, with the help of its dedicated libraries, have the same kind of support as JavaScript with its BaconJS.

References

[1] A History of Haskell – Being Lazy With Class, P. Hudak, 2007
[2] A Survey of Functional Reactive Programming, E. Amsden, 2013
[3] An Axiomatic Semantics for Functional Reactive Programming, C. King, 2008
[4] An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications, K. Kambona, 2013
[5] Arrows, Robots, and Functional Reactive Programming, P. Hudak, 2003
[6] Deprecating the Observer Pattern, I. Maier, 2010
[7] Directing JavaScript with Arrows (Functional Pearl), K.Y. Phang, 2010
[8] Event-Driven FRP, W. Taha, 2002
[9] Flapjax – Functional Reactive Web Programming, L. Meyerovich, 2007
[10] Frappe – Functional Reactive Programming in Java, A. Courtney, 2001
[11] FRP in C++, X. Dai, 2010
[12] Functional JavaScript, M. Fogus, 2013
[13] Functional Reactive Animation, C. Elliott, 1997
[14] Functional Reactive Programming, S. Blackheath 2015
[15] Keeping Calm in the Face of Change, Towards Optimisation of FRP by Reasoning about Change, N. Sculthorpe, 2011
[16] Liftless Functional Reactive Programming, C. Monsanto, 2009
[17] Multi-tier Functional Reactive Programming for the Web, B. Reynders, 2014
[18] Parallel Functional Reactive Programming, J. Peterson, 2000
[19] Practical Functional Reactive Programming, J. Peterson, 2014
[20] Programming Paradigms for Dummies – What Every Programmer Should Know, P. Van Roy, 2009
[21] Real-Time FRP, Z. Wan, 2001
[22] The C++ Programming Language 3rd. ed., B. Stroustrup, 1997
[23] The Conception, Evolution, and Application of Functional Programming Languages, P. Hudak, 1989
[24] The Yampa Arcade, A. Courtney, 2003
[25] Towards a Declarative Web, A. Ekblad, 2012
[26] Why Functional Programming Matters, J. Hughes, 1990
[27] Wormholes – Introducing Effects to FRP, D. Cort, 2012
[28] http://javascript.crockford.coml
[29] JavaScript: The Good Parts, D. Crockford, 2008
[30] Asynchronous Functional Reactive Programming for GUIs, E. Czaplicki, 2013
[31] schlangster.github.io/cpp.react
[32] www.flapjax-lang.org
[33] Reactive Web Applications with Dynamic Dataflow in F#, A. Tayanovskyy, 2014
[34] http://elm-lang.org
[35] http://haskell.cs.yale.edu
[36] https://wiki.haskell.org/Yampa
[37] https://baconjs.github.io
[38] Mozilla Developer Network (MDN) https://developer.mozilla.org

About the author

I am a Software Developer with a few years of experience with C++, JAVA, PHP, and JavaScript. I work in the MBB Single RAN OMCP department that develops BTS's main OAM component in the JavaScript language with an FRP style on a NodeJS framework.

Bartosz Kwaśniewski
Software Developer
MBB Single RAN

Python: A General-purpose Language with a Low-level Entry Barrier

Bartosz Woronicz
Engineer, Software Configuration
MBB System Module

NOKIA

Introduction

An early idea of the Python programming language dates back to the late 1980s. At that time, a talented Dutch programmer Guido van Rossum had already gained quite a lot of experience with the ABC programming language, which later on significantly influenced his subsequent and a more mature creation, the so-called Python. The programming language inspired by ABC, which van Rossum created, consists of the following principles:

- **Dynamic**: variable declarations are not required, in contrast to C/C++ where you have to specify, declare the type of the data, i.e. `int count; char keystroke;` etc. In Python, we simply use `var = "somevalue"`
- **Strongly typed**: one cannot implicitly add two data objects of different types, see the Python interpreter example:

```
>>> message = "I like cheese and number"
>>> favnum  = 7
>>> message + favnum
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

- **Statement nesting by indentation**: the so-called off-side rule [1]. Another example in Python:

```
class MyClass(object):

    def mymethod(self, *args, **kwargs):
        execution_block()

    def print_name(self):
        print self.name
```

- **Infinite arithmetic precision**: the possibility to use unrestricted real numbers, which is particularly convenient for novices at the language

The Python language has been positively appraised by a number of companies, organizations, and governmental institutions worldwide, which has significantly contributed to its widespread usage within Nokia.

1. Python@Nokia Networks

Every large company like Nokia annually faces the issue of employees switching projects internally and new people being hired. This means that a code which has been written by one person must now be maintained and extended by others. In point of fact, this is Python's most valuable advantage, a strong emphasis on code's readability. By utilizing strict rules, it enforces a uniform style; for instance, as a result of the indentation rule, the code has an appropriate and unchanging visual structure. Moreover, on their website, the Python Foundation (an organization managing the language's development) offers a series of documents known as Python Enhancements Proposal (PEP); they are design documents for confronting new major features of the language with the programming community. In addition, there is a document which provides a thorough coding style guide, like the very well-known and widely acclaimed PEP8 [2]. In a way, it is reminiscent of *Request for Comments* (RFC), a document detailing standards in the world of computer network connectivity.

The language is truly general in its purpose, starting from a simple script hacking to Web applications (webframeworks *Django*, *Flask*) and scientific applications (*SciPy*, *pandas*, *matplotlib*), a version control system (*mercurial*), and even a machine-learning library (scikit-learn). In Software Configuration Management (SCM), where I work, it has been utilized for an automation machine with the Continuous Integration system as well as for writing certain internal tools such as web applications for gathering and processing build data.

As it has been mentioned in the Introduction, Python has become particularly famous for its suitability for young apprentices seeing as it is probably one of the easiest languages to immediately start coding with. New coders can easily acquire the most common parts, then go deeper and optimize their approach by incrementally gaining new skills. This language is multi-paradigmatic, so one can move from using imperative, procedural operations such as *for-loop* to proceed then to *list-comprehension*, which is typical for functional languages (see **Listing 1**).

Listing 1

```
# this is simple for-loop building the list of squares of the
elements from 0 to 9
new_list = []
for i in range(10):
    new_list.append(i**2)

# this is list comprehension
new_list = [i**2 for i in range(10)]
```

The example below depicts a transformation into a more concise code string as well as the acceleration of execution. Let us run the test with a *timeit* module directly from the console (see **Listing 2**).

Listing 2

```
$ python -mtimeit 'new_list = []' 'for i in range(10): new_list.
append(i**2)'
1000000 loops, best of 3: 1.56 usec per loop

$ python -mtimeit 'new_list = [i**2 for i in range(10)]'
1000000 loops, best of 3: 0.931 usec per loop
```

It is evident from the example in **Listing 2** that the second runtime is significantly shorter.

2. Scripting language

Python has an interpreter. It has been designed to interact with the user. As a result, it is very convenient for prototyping. Providing that it is a scripting language, it allows its internal behavior to be tested and modified by the interpreter. An interactive interpreter can be used to test the Python code ad hoc.

XML files are very common for the kind of work which is done within SCM. They contain a lot of information about the software development. Getting the code parsing XML right the first time around might not be easy; however, with a Python interpreter shell, one can step-by-step create a working prototype with it.

An example of XML file, similar to the one in **Listing 3**, is used to provide a mapping for the software release note in the bugtracker software with a list of regular expressions for new features, changes, etc.

Listing 3

```
<?xml version="1.0" ?>
<RN-mapping version="1.1">
    <configuration>
        <separator scripts="yes">
            <item regexp="(?=[: ])"/>
        </separator>
    </configuration>
    <mapping>
        <correctedFaults scripts="yes" >
            <item BUGTRACKER_ID="CF" BUGTRACKER_Type="Fault"
                  regexp="(?&lt;=CF[ ])[A-Z0-9]+|CF[0-9]+"/>
        </correctedFaults>
        <revertedCorrectedFaults scripts="yes" >
            <item BUGTRACKER_ID="CF" BUGTRACKER_Type="Withdrawal"
                  regexp="(?&lt;=CF[ ])[A-Z0-9]+|CF[0-9]+"/>
        </revertedCorrectedFaults>
        <features scripts="yes" >
            <item BUGTRACKER_ID="NF" BUGTRACKER_Type="Feature"
                  regexp="NF-[0-9]+"/>
            <item BUGTRACKER_ID="NFI" BUGTRACKER_Type="Feature"
                  regexp="NFI\s?[0-9._]+"/>
            <item BUGTRACKER_ID="FCR" BUGTRACKER_Type="Feature"
                  regexp="FCR\s?[0-9._]+"/>
        </features>
        <changenotes scripts="yes" >
            <item BUGTRACKER_ID="NF" BUGTRACKER_Type="Feature"
                  regexp="NF-[0-9]+"/>
            <item BUGTRACKER_ID="NFI" BUGTRACKER_Type="Feature"
                  regexp="NFI\s?[0-9._]+"/>
            <item BUGTRACKER_ID="FCR" BUGTRACKER_Type="Feature"
                  regexp="FCR\s?[0-9._]+"/>
        </changenotes>
        <unsupportedFeatures scripts="no" >
            <item BUGTRACKER_Type="???"/>
        </unsupportedFeatures>
        <restrictions scripts="no" >
            <item BUGTRACKER_Type="Restriction"/>
        </restrictions>
        <neededConfigurations scripts="no" >
            <item BUGTRACKER_Type="???"/>
        </neededConfigurations>
    </mapping>
</RN-mapping>
```

In the interactive shell session below, we parse the XML file with an XPath query language, utilizing Python's *xml* module (see **Listing 4**). First we get the root element, and then we search for the node under the **mapping** node called **features**, with a **scripts** attribute with a *yes* value. In this way, we get a list of features of the new release note, and we fetch the **regexp** attribute value when parsing it.

Listing 4

```
>
>> from xml import etree as ET
>>> tree = ET.parse('/tmp/XML_RN_mapping.xml')
>>> root = tree.getroot()
>>> print root
<Element RN-mapping at 0x7fecb5021b00>
>>> root.xpath('mapping/features[@scripts="yes"]/*')
[<Element item at 0x7fecb5021f80>,
 <Element item at 0x7fecb5021fc8>,
 <Element item at 0x7fecb5029050>]
>>> [x.attrib['regexp'] for x in root.xpath('mapping/features[@
scripts="yes"]/*')]
['NF-[0-9]+', 'NFI\\s?[0-9._]+', 'FCR\\s?[0-9._]+']
>>>
```

3. Object-oriented language

In contrast to other popular object-oriented programming (OOP) languages, in Python literally everything is object-oriented. The applied approach contains typical OOP features such as inheritance and polymorphism, but excluding access control in methods (e.g. *private, protected, public* in *C++/Java*). The justification for this is provided by the motto "We're all adults here," as the creator has decided that a feature is an unnecessary bloat. There is only a convention to express the private method in code; it consists in preceding the name of the method with a double underscore. In such a case, it only means that the method is intended to run internally, not directly. Moreover, this will trigger the mechanism to internally rename the method from **__method** to **_classname__method** to avoid name clashing in the classes inheriting [4]; see the example in **Listing 5** below.

Listing 5

```
class Foo(object):
    def update(self, **kwargs):
        self.attrs = kwargs
    __update = update

class Bar(Foo):
    def update(self, **kwargs):
        self.attrs = kwargs * 2
```

- **It is (quite) fast**
  In typical scripting languages such as Bash and other Shell variants, the code is processed line by line or rather by a tokenization process. By means of language processing, the code fragments are tokenized, i.e. assigned to categories, and then their interaction is processed lexically.
  In Python, code is compiled to its virtual machine (intermediary) code, and then executed. In this way, the program's execution becomes much faster. The memory is handled by a garbage collector, so it does not need to be freed manually. However, in tasks, where the execution time is critical, some libraries, classes, and methods can be written in C/C++ and provide Python bindings so that part of code runs blazingly fast. The examples are *lxml* library, *OpenSSL* library, and many others.

- **Portability**
  The Python code can be launched on any operating system and processor architecture that has compiled the Python interpreter. As an example, the automation scripts that we create run on both *Microsoft Windows* and *GNU Linux* hosts, running the building processes for a different generation of software without any compilation (in contrast to C/C++ or Java). Nevertheless, in some cases, scripts have to handle differences between operating systems as is the case in the example in **Listing 6**.

Listing 6

```
import os
if os.name == 'nt':
    tool_regexp = 'CreateBinary[^.]*[.]exe'
else:
    tool_regexp = 'CreateBinary[^.]*[.]linux'
…
```

- **Play with classes**
  Now, we shall move on to a more sophisticated example.
  Let us create a stub class called **MyClass**, and then define its function as **print_with_stars**. Next, it is necessary to attach a method to the already created class. It is quite clear that the method is provided by the class' internal dictionary (see **Listing 7**).

Listing 7

```
>>> class MyClass(object):
...     pass
...
>>> def print_with_stars(self):
...     print "*%s*" % self
...
>>> import types
>>> MyClass.print_with_stars = types.MethodType(print_with_
stars)
>>>
>>> MyClass.__dict__
dict_proxy({'__dict__': <attribute '__dict__' of 'MyClass' ob-
jects>, '__module__': '__main__', '__weakref__': <attribute '__
weakref__' of 'MyClass' objects>, '__doc__': None, 'print_with_
stars': <function print_with_stars at 0x7fe81ca50668>})
```

However, in this way, the function attached will not become a method. This so-called monkey-patching is wrong because it does not change the existing instances. Imagine a situation where we have already created a **mc** object, and then we attached the function to its class (see **Listing 8**).

Listing 8

```
>>> mc = MyClass()
>>> mc2 = MyClass()
>>> MyClass.print_with_stars = print_with_stars
>>> mc.print_with_stars
<bound method MyClass.print_with_stars of <__main__.MyClass
object at 0x7fa0db2e34d0>>
>>> mc2.print_with_stars
<bound method MyClass.print_with_stars of <__main__.MyClass
object at 0x7fa0dc1ce3d0>>
```

Thus all the class instances will have this method bounded.

Instead, the *types* [3] should be used to modify the created class-adding method appropriately (see **Listing 9**).

Listing 9

```
>>> import types
>>> mc = MyClass()
>>> mc2 = MyClass()
>>> mc.print_with_stars = types.MethodType(print_with_stars, mc)
>>> mc.print_with_stars
<bound method ?.print_with_stars of <class '__main__.MyClass'>>
>>> mc2.print_with_stars
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute 'print_with_
stars'
```

- **Useful modules and tools**
  The great power of Python is the community. The community is building and maintaining a number of tools and libraries that can be reused for many different applications. The most straightforward access to them is to install them with a program called PIP.
  Of course, there are also tools such as *easy_install*, *setuptools*, etc. In this paper, however, we will focus specifically on PIP.

  - **PIP**
    This tool enables access to the database of 3rd party Python modules, together with its dependencies. It can be installed locally for a particular user. On the Unix-like systems (Linux, BSD, etc.), you may add *~/.local/bin* to your local *PATH* variable, and then download and install the tool by scripts (see **Listing 10**).

Listing 10

```
$ wget -O- https://bootstrap.pypa.io/get-pip.py | python – --user
```

  - **Ipython**
    It is an extremely popular tool [5]. It provides an extended Python interpreter with an easy completion by a Tab-key similar to the one in the Unix-like system terminal. It covers many extra features, including a webapplication notebook for accessing an interactive shell in a webrowser with extra features such as printing charts and graphics instantly, which is impossible in the regular console.

  - **Flask**
    The Python's world is vast; if we wanted to create a simple webapp listening on port 8080, for instance, the Flask micro-webframework [6] might be used for that purpose. Here is an example of a simple application with a route to URL */api* (see **Listing 11**).

Listing 11

```
from flask import Flask, jsonify
app = Flask(__name__)

data = {
    "name": Bartosz,
    "surname": "Woronicz",
    "age": 29
}

@app.route("/api")
def hello():
    return jsonify(data)

if __name__ == "__main__":
    app.run()
```

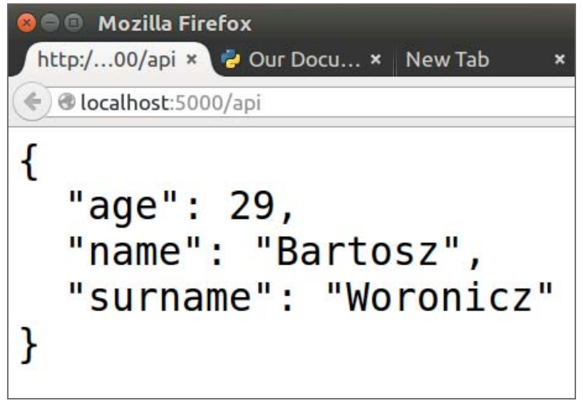
Now, we can install the necessary package with the previously mentioned PIP and run it (see **Listing 12**).

Listing 12

```
$ pip install --user Flask
$ python hello.py
 * Running on http://localhost:5000/
```

And when we open the browser, we can see the outcome (see **Figure 1**).

Figure 1 The output from Flask application.

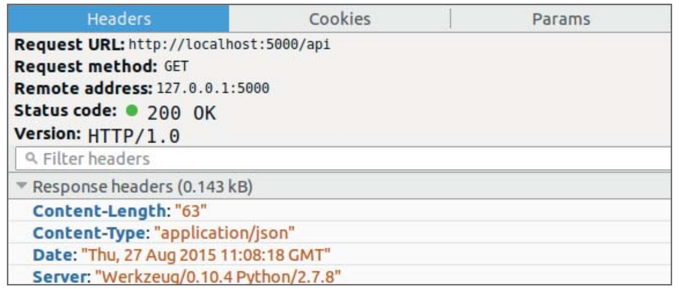


```
{
  "age": 29,
  "name": "Bartosz",
  "surname": "Woronicz"
}
```

When we take a look at HTTP protocol headers, the response has the *Content-Type* property set to *application/json* (see **Figure 2**).

  - **requests**
    The *requests* module is probably the best of the 3rd party modules ever created for Python. For instance, communication with an API webapplication is quite easy, as it can be seen in **Listing 13**; a post method is being used with a data parameter, the dictionary.

Figure 2 HTTP headers from Flask applications.



Request URL: http://localhost:5000/api
Request method: GET
Remote address: 127.0.0.1:5000
Status code: ● 200 OK
Version: HTTP/1.0

Response headers (0.143 kB)

Content-Length: "63"
Content-Type: "application/json"
Date: "Thu, 27 Aug 2015 11:08:18 GMT"
Server: "Werkzeug/0.10.4 Python/2.7.8"

Listing 13

```
import requests
data = {
    "name": "Guido",
    "surname": "van Rossum"
}
requests.post('http://myapplication/api/user/add', data=data)
```

4. Python 3: the next generation

As for Python 2.0 the giant leap was the introduction of a better, full-garbage collection and Unicode support, Python 3 breaks compatibility quite a lot. Some features of Python 3 were already included in Python 2.7.x, but they were optional; for example, the new way of string formatting (see **Listing 14**). Here is a list of the most interesting ones. It is also worth mentioning that the new behavior and syntax can be forced in 2.x by using a special *__future__* module [7].

Listing 14

```
# Python 2.x code
from __future__ import print_function
print("Hello Nokia!")
```

Generators

In the 3.x edition, most functions that were returning a *list* type, now they are returning a generator object. For example, instead a special built-in function **xrange()**, now **range** (returning the list of integers by amount, start, stop, and step between) acts in the same way as **xrange** in the older Python. Apart from dictionary **d**, the method **d.items()** will also return a generator.

Print function

Starting from Python 3, *print* is no longer an instruction; it is a built-in function (see **Listing 15**).

Listing 15

```
# Python 2.x code
print "Hello Nokia!"
# Python 3.x code
print("Hello Nokia!")
```

- **Unified int**
  Now **long** and **int** types become one **int** type, but in most cases it behaves like an old **long** type.

- **Arithmetic operations**
  A division of two integers will give a float instead of being truncated into an integer. As a result, **1/2** in Python 2.x produces **0**, and in version 3.x it produces **0.5**.

5. Integration with other languages

Python can be easily integrated with other languages. It works both ways, by embedding (Python as a library for C) or extending (C library used in Python). The main implementation is written in C/C++ code (so-called CPython). Moreover, there are alternative implementations such as PyPy (written in Python itself), Cython (generating static C code), Jython (Java) or IronPython (Python for .NET framework).

6. Drawbacks

As is the case with every new programming language, the newcomers to the world of Python are having a hard time understanding some of the language's complexities. The language seems easy at first, but the coders encounter difficulties because they misunderstand certain concepts. Python's Wiki webpage [8] provides examples of such errors, together with links to related articles. The list is long, but the greatest obstacle seems to be that coders who have previously used C/C++ try to apply its principles to Python as well. A few examples of the most common errors are as follows:

- Do not use parentheses around test, e.g. **if (x=="something"):**, just **if x=="something":**
- You cannot use assignments in while loop tests, e.g. **while ((x=next() != NULL))**. But this is a design decision because it is a common mistake in other languages to use a comparison operator instead of an assignment and vice versa.
- Over complication while working with iterables, e.g. see **Listing 16**.

Listing 16

```
# the given string "cheeseshop"
mystring = "cheeseshop"
# instead of trying something like
for i in range(len(mystring)):
    print mystring[i]
# or even
i = 0
while i < len(mystring): print mystring[i]; i += 1
# just iterate over the mystring
for c in mystring: print c
```

- Do not expect from function changing objects to return something other than **None**; it is an extremely common mistake for a novice lacking an understanding of the difference between mutable and immutable objects. The list is mutable, so it can be changed in-place without creating a copy, so the result of **mylist.append(X)** is just **None**. But now **mylist** contains element **X**. String, on the other hand, is immutable, and the operation **"smart" + " fox"** produces **"smart fox"**, which is the new string object.

Conclusion

This paper was meant as a brief introduction to Python. On the Internet, there is a lot of literature, use cases, and a number of examples how Python can be utilized if one felt the need to examine the matter thoroughly. This programming language is easily applicable, well-documented, and works well even for people completely unfamiliar with coding. Python may not be appropriate for certain use cases, but for most tasks its versatility is simply ideal.

References

[1] https://en.wikipedia.org/wiki/Off-side_rule
[2] https://www.python.org/dev/peps/pep-0008/
[3] https://docs.python.org/2/library/types.html
[4] https://docs.python.org/2/tutorial/classes.html#private-variables-and-class-local-references
[5] http://ipython.org/
[6] http://flask.pocoo.org/
[7] https://docs.python.org/2/library/__future__.html
[8] https://wiki.python.org/moin/BeginnerErrorsWithPythonProgramming

About the author

I work at MBB System Module (SCM), where I am responsible for maintaining smooth operation of the software-building machine. This is accomplished with a number of different tools and scripting glue, written primarily in Python and shell scripts, that connects them. I gather VCSes (version control systems), continuous integrations, and tests of software in one lined-up mechanism. On the one hand, we enjoy it very much when the software-building process picks up speed and everything goes according to plan; on the other hand, we never hesitate to go overdrive in case of a sudden disaster. Additionally, I create and co-create internal and external training courses and workshops on Python.

Bartosz Woronicz
Engineer, Software Configuration
MBB System Module

Beginning the Adventure: Writing a Minimal Compiler

Michał Bartkowiak
Engineer, Software Development
MBB FDD LTE

Why would anyone want to write another compiler when there are so many of them around? For example, sometimes it is beneficial to create and use a domain-specific language (DSL) to facilitate the development process (e.g. testing). But before the creation of a production-quality language and its compiler, necessary knowledge and experience has to be gathered. This article is a practical introduction to development of compilers through providing a basic example of a programming language and its compiler.

1. Prerequisites

1.1. An overview of the compilation process

Compiler is a computer program that transforms a source language into the target language. Common example of such transformation is compilation of C++ source code into x86-64 assembly. In such case the compiler's input is a set of text files and resulting output is an executable.

Compilation process can be divided into following stages:

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Optimization
- Code generation

Lexical analysis divides source code into tokens, which can be understood as words of the source language. Parser, provided with the tokens, performs syntax analysis. In other words, it validates the structure of the code. As a result, Abstract Syntax Tree (AST) is generated.

The aim of semantic analysis is to comprehend the meaning of the program. For non-trivial programs this is an extremely hard task [1]. Thus compilers perform only limited amount of this analysis to catch inconsistencies, e.g. usage of undefined variables or type incompatibility in assignments.

Optimization is a transformation of program code in result of which program runs faster or uses less of expensive resources (e.g. memory or size). This modification cannot change the semantics of the program.

Finally, given all information calculated in previous stages, code generation produces code in target language.

1.2. MiNiK language

To demonstrate what is needed to create a basic compiler, simplistic language is used. The MiNiK language has the following properties:

- In its global scope only functions are allowed.
- The only data type is a 4-byte integer.
- A function takes a fixed number of arguments and returns a single value.
- Arithmetic expressions and comparisons are allowed.
- Functions can only be called with variables as arguments.
- There is one conditional statement (**`if`**) and one loop (**`while`**) available.
- **`read`** and **`write`** functions are provided as a standard library.
- A function called **`main`** has to be defined in the source code by the programmer as an entry point to the program

Short example of MiNiK code is shown in **Listing 1**.

Listing 1 Example of MiNiK code: factorial calculation.

```
function factorial(val) {
  if (val == 0) {
    return 1
  }
  var newVal := val - 1
  return val * factorial(newVal)
}

function main() {
  var val := read()
  return factorial(val)
}
```

This article is accompanied by MiNiK compiler's source code available at Nokia Book's Github [2]. In order to fully benefit from the presented material, source code and associated comments should be analyzed in parallel with the article. Makefiles included in the MiNiK's compiler source code should serve as a guide for building various modules of the software.

2. MiNiK Compiler

2.1. LEX: Division of input into tokens

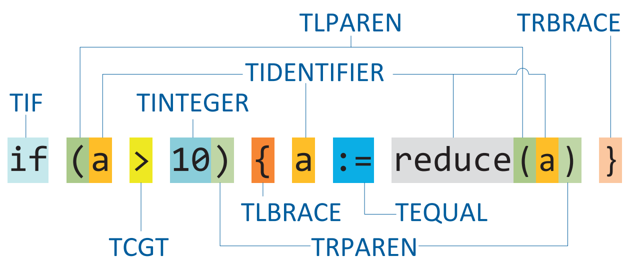
Lexing process is conducted as a left-to-right scan of input string constituting the source code. The most commonly used tools for performing the lexical analysis are regular expressions [3]. Since lexing phase is usually similar among various compilers, dedicated software to facilitate this task has been developed. The Fast Lexical Analyzer (flex) [4] is one of them. To generate a lexer with flex, it is sufficient to create a file with set of pairs of regular expressions and C code. Such pair is called a rule. Selected lexing rules for MiNiK are shown in **Listing 2**.

Listing 2 Selected lexing rules for MiNiK.

```
"function"             return TOKEN(TFUNCTION);
"return"               return TOKEN(TRETURN);
"if"                   return TOKEN(TIF);
"while"                return TOKEN(TWHILE);
"var"                  return TOKEN(TVAR);
[a-zA-Z_][a-zA-Z0-9_]* SAVE_TOKEN; return TIDENTIFIER;
[0-9]+                 SAVE_TOKEN; return TINTEGER;
":="                   return TOKEN(TEQUAL);
"("                    return TOKEN(TLPAREN);
[...]
```

Tokens extracted by flex are then passed consecutively to next phase. Example of lexing of MiNiK's code fragment is shown in Figure 1.

Figure 1 Example of lexing of MiNiK's code fragment.



2.2. YACC: A tool for building the Abstract Syntax Tree

Given the tokens from lexing stage, syntax rules are checked and AST is generated by the parser. Similarly to lexing, this phase is often done with the help of dedicated software (parsers generators), called Yet Another Compiler Compiler (YACC). One of the most commonly used today is GNU Bison [5]. Bison can generate parser for a language, if it is described by context-free grammar (CFG) [3] represented in Backus-Naur Form (BNF) [3]. Simply speaking, Bison file contains a set of rules for constructing syntactic groupings using tokens and/or other syntactic groupings [5] supplemented with semantic actions defined as a C or C++ code. Listing 3 shows selected Bison rules for MiNiK parsing. C++ actions have been omitted as they will be described in details in next section.

Listing 3 Selected parsing rules for MiNiK.

```
Program : Functions { ... };

Functions : Function { ... }
  | Functions Function { ... };

Identifier : TIDENTIFIER { ... };

ExpressionMult : ExpressionAtom { ... }
  | ExpressionAtom MultOp ExpressionMult { ... };

ExpressionAdd : ExpressionMult { ... }
  | ExpressionMult AddOp ExpressionAdd { ... };

Expression : ExpressionAdd { ... }
  | ExpressionAdd CompOp ExpressionAdd { ... };

AssignStatement
  : Identifier TEQUAL Expression { ... };

IfStatement
  : TIF TLPAREN Expression TRPAREN Block { ... };

[...]

AddOp  : TPLUS | TMINUS;
MultOp : TMUL | TDIV;
CompOp : TCEQ | TCNE | TCLT | TCLE | TCGT | TCGE;
```

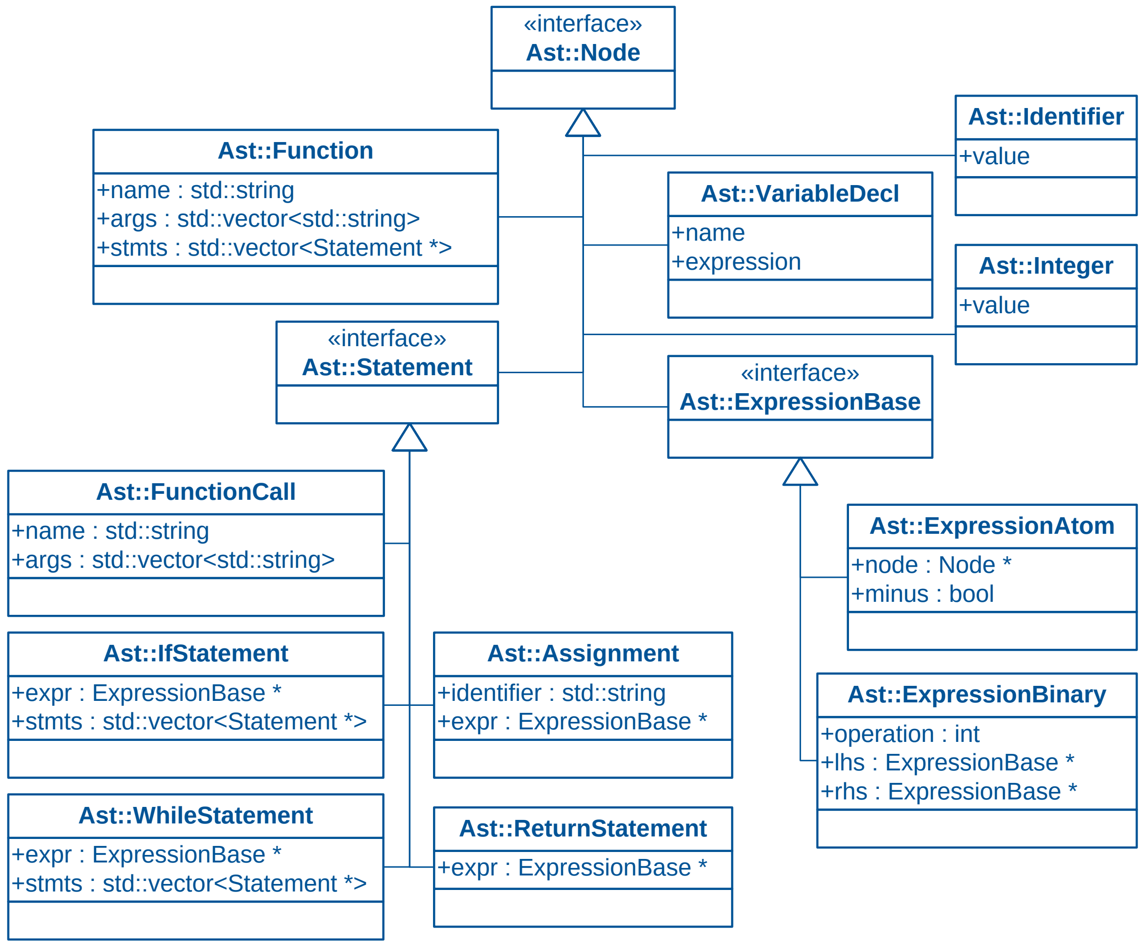
Given these rules, generated parser is able to, for example, recognize **`AssignStatement`** when it encounters an **`Identifier`** followed by assignment token (**`TEQUAL`**) and an **`Expression`**.

When writing BNF rules, there is often a problem with conflicts existing inside the grammar. Example of this problem is handling the priority and associativity of operators. In Bison it could be solved by defining operators' precedence but in MiNiK another approach has been adopted. Rules for expressions are built hierarchically and it can be noticed that **`ExpressionMult`** will be earlier selected (or reduced in CFGs terminology) by generated parser than **`ExpressionAdd`** (i.e. with higher priority).

2.3. Abstract Syntax Tree

Syntax tree is called abstract because during its creation only information necessary for further processing is used. It means that e.g. information about keywords or parentheses can be omitted because it is implicitly embedded in the tree. The level of abstraction is determined by the design of hierarchy of tree's nodes. Diagram of the nodes in MiNiK compiler is shown in Figure 2.

Figure 2 Hierarchy of nodes of MiNiK's abstract syntax tree.



MiNiK program consists of a set of functions and each function is represented by one AST. Consequently, whole program is represented as a set of ASTs. Note that hierarchy of lexical scopes is also encoded in the AST structure.

The ASTs themselves are created during the execution of semantic actions of Bison rules. Semantic actions for selected parsing rules are shown in Listing 4. For example, when generated parser encounters **`TIDENTIFIER`** token new AST node **`Ast::Identifier`** is created. Similarly, more complex actions are constructed. The **`Ast::FunctionCall`** node is constructed from **`Ast::Identifier`** and **`Ast::Arguments`**. At the same time, **`TLPAREN`** and **`TRPAREN`** tokens (parentheses) are simply ignored as superfluous for the AST.

Creation of unnecessary levels of AST is also avoided. For example in the first rule of **`Expression`** the **`ExpressionAdd`** is just forwarded as a result.

Listing 4 Selected parsing rules for MiNiK with their semantic actions.

```
Program
  : Functions { program = *$1; delete $1; };

Identifier
  : TIDENTIFIER
    { $$ = new Ast::Identifier{*$1}; delete $1; };

Statements
  : Statement { $$ = new Ast::Statements{$1}; }
  | Statements Statement { $1->push_back($2); };

FunctionCall
  : Identifier TLPAREN Arguments TRPAREN
    { $$ = new Ast::FunctionCall{$1->val, *$3};
      delete $1; delete $3; };

Expression
  : ExpressionAdd { $$ = $1; }
  | ExpressionAdd CompOp ExpressionAdd
    { $$ = new Ast::ExpressionBinary{$1, $2, $3}; };
```

Listing 5 shows conditional statement represented as a fragment of AST. Two main parts can be isolated: binary comparison **Ast::ExpressionBinary**: '**a > 10**' and **Ast::Assignment**: '**a := reduce(a)**'. Further analysis of **Ast::Assignment** shows that result of call of '**reduce**' function is assigned to variable **a**.

Note that tokens like '**if**', '**(**', '**)**' '**{**', '**}**' or '**:=**' are not represented explicitly in the AST. Their presence can be derived from the type of AST's nodes or AST's structure.

Listing 5 Fragment of AST which represents following MiNiK code:

```
if (a > 10) { a := reduce(a) }

<IfStatement>
    <ExpressionBinary>
        <ExpressionAtom>
            <Identifier> 'a'
        <Operation> '>'
        <ExpressionAtom>
            <Integer> '10'
    <Assignment> 'a'
        <ExpressionAtom>
            <FunctionCall> 'reduce'
                <Arguments> 'a'
```

2.4. Semantic analysis

After the AST is created, compiler is able to run *semantic analysis*. Various kinds of checks can be performed during these stages, depending on the compiled language [3]. For example:

- Checking whether all functions and variables are declared only once in given scope.
- Type checking.
- Performing advanced static analysis, e.g. available expressions analysis [1].

MiNiK's source code is checked if:

- all identifiers are unique.
- all used identifiers (functions and variables) are defined.

As a first step, MiNiK compiler is iterating over **Ast::Function** nodes to gather information about functions. Their names and numbers of arguments are remembered. If any name is duplicated, an error is raised.

In second stage AST is visited again. For **Ast::FunctionCall** nodes it is verified whether:

- a function with given name is defined.
- the number of arguments of function call is the same as in the function declaration.

Additionally, during top-down traverse through the AST, lexical scopes are visited according to their nesting hierarchy. That is why checks for variables' declarations and usage are performed correctly.

2.5. Execution and memory model

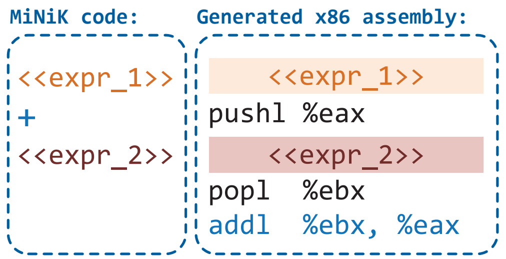
Before explaining how the code generation works, the execution and memory model of compiled MiNiK programs has to be presented.

As a fundamental convention, it is established that every expression in MiNiK leaves its result in **EAX** register. Next, the mechanism of function call has to be introduced. Firstly, activation record (frame) [3] is created on the stack. It consists of function's arguments, return address and space for function's local variables. The size of activation record is proportional to the number of arguments and maximum number of local variables available at any point of function execution. Current value of variable is always stored in dedicated location in memory.

At exit, function's return value is passed via **EAX** register, as in case of every expression. Additionally, the space allocated on the stack for the local data is released.

In advanced compilers, as many variables as possible are held in registers. To achieve this, specialized algorithms for register allocation are used, based on e.g. graph coloring problem [3].

Figure 3 Code generation scheme for composing of MiNiK's expressions.


MiNiK code:

```
<<expr_1>>
+
<<expr_2>>
```

Generated x86 assembly:

```
    <<expr_1>>
pushl %eax
    <<expr_2>>
popl  %ebx
addl  %ebx, %eax
```

2.6. Code generation

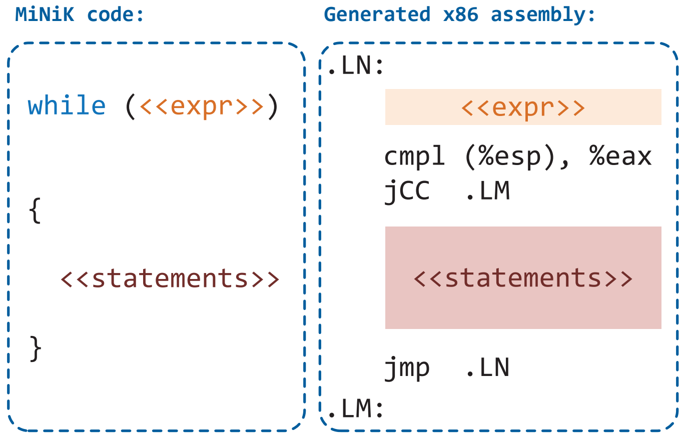
Translation to assembly code is the final step of MiNiK compilation process. To make it universal and possibly close to real-world scenarios, MiNiK compiler generates x86 assembly [6]. It is produced by visiting AST nodes recursively by **CodeGen** class.

MiNiK language constructs are translated to assembly blocks. A block is a list of assembly instructions and, if necessary, a label is assigned to the block to allow making the references needed for jumps. All generated blocks are arranged into a sequence which represents compiled MiNiK program.

First phase of compilation is traversal of all ASTs in order to collect information about functions. Number of their arguments is remembered and size of activation record is calculated. This calculation is done during the traversal of function's AST. Additionally, functions' names are saved and symbolic functions' names are generated. Symbolic names differ from original names in order to avoid collisions with the already used names in system libraries. Second phase is also a top-down traversal of ASTs, but this time assembly is produced.

Expressions are basic building blocks in MiNiK. Since they can be composed to create another expression, code generation process has to be able to build arbitrary levels of nested expressions. What helps now is the convention which tells that every expression has to leave its resulting value in **EAX** register. This way, scheme shown in **Figure 3** is enabled. The scheme can be nested, treating every expression as a black box.

Figure 4 Code generation scheme for MiNiK's while statement.


MiNiK code:

```
while (<<expr>>)

{

  <<statements>>

}
```

Generated x86 assembly:

```
.LN:
      <<expr>>

    cmpl (%esp), %eax
    jCC  .LM

      <<statements>>

    jmp  .LN
.LM:
```

Translation of functions is done according to a memory model. Firstly, generated assembly reserves space for local data (part of activation record). Calculation of offsets for function's arguments and local variables is facilitated by **StoreManager** class. **StoreManager** associates memory cell with a variable, if and only if it is visible in a currently compiled scope. As a next step, every statement of the function is translated. At the end return value is stored in **EAX** register and activation record is released and control is returned to caller's site.

Next element worth attention is the **while** loop. It is translated in three main steps, supplemented by appropriate labels, comparison and jumps:

- Translation of expression which forms loop condition
- Translation of condition itself
- Translation of while block statements

Conditional (**if**) statements are translated analogously to **while** statements with one difference: looping jump (**jmp .LN**) and its corresponding label are not needed.
Generation of assembly for function call comprises of three steps:

- Pushing arguments onto the stack in conformance with memory model
- Calling the function via **call** instruction
- Releasing the stack space occupied by arguments passed to function

As a wrap-up of this section, **Listing 6** presents fragment of MiNiK's code with corresponding assembly code.

Listing 6 Assembly generated for simple MiNiK program.

```
                         .text
                         .global __MINIK_main
function inc(a)        __MINIK_inc:
{                        enter  $0, $0
  return a + 1           movl   8(%ebp), %eax
}                        pushl  %eax
                         movl   $1, %eax
                         addl   (%esp), %eax
                         leave
                         ret
function main()        __MINIK_main:
{                        enter  $4, $0
  var a := 1             movl   $1, %eax
                         movl   %eax, -4(%ebp)
                       .L0:
  while (a == 1)         movl   -4(%ebp), %eax
  {                      pushl  %eax
                         movl   $1, %eax
                         cmpl   (%esp), %eax
                         jne    .L1
    a :=                 movl   -4(%ebp), %eax
                         pushl  %eax
         inc(a)          call   __MINIK_inc
                         addl   $4, %esp
                         movl   %eax, -4(%ebp)
  }                      jmp    .L0
                       .L1:
  return a               movl   -4(%ebp), %eax
                         leave
}                        ret
```

2.7. Runtime environment

Minimalistic runtime environment and standard library has been created, see the **`minikrt.c`** file.

The task of runtime environment is to call the MiNiK's main function (which symbol name is **`__MINIK_main`**) and print the result it returns. Standard library is composed of two functions: **`read`** and **`write`**. The first function allows reading integer number from standard input. The second one writes passed argument to standard output.

3. Next steps

Both MiNiK language and its compiler can be further developed. This could be a great exercise to someone who would like to gain some practice in writing of a compiler.

For example, the language lacks the following features:

- Possibility of passing expressions as function's parameters
- **`else-if`** and **`else`** conditional statements
- Other kinds of loops (**`do-while`**, **`for`**)
- Global variables or constants
- Other data types, e.g. double or string
- Pointer types
- Exceptions
- More extensive standard library

In the compiler, the following features can be implemented in the first place:

- Register allocation algorithm
- Basic optimizations, e.g. avoiding of unnecessary pushes or comparisons
- Generation of intermediate code during compilation, e.g. *three-address code*

4. Conclusion

Hopefully, after reading this article, you are familiar with basics of compiler's construction. However, you have to remember that described compiler, although fully-functional, is a very simplistic one and MiNiK language lacks elementary functionalities. However, diving into MiNiK's code, experimenting with it and implementing new features could be the beginning of Quest for Dragons [7].

Resources

[1] Nielson F., Nielson H. R., Nielson, C. H., Principles of Program Analysis, Springer, 2004.

[2] Nokia Book's source code repository, Nokia. Available online: https://github.com/nokia-wroclaw/nokia-book.

[3] Aiken A., Stanford's Compilers. Available online: https://class.coursera.org/compilers-004.

[4] flex: The Fast Lexical Analyzer. Available online: http://flex.sourceforge.net/.

[5] GNU Bison – The Yacc-compatible Parser Generator. Available online: https://www.gnu.org/software/bison/manual/.

[6] Intel® 64 and IA-32 Architectures Software Developer Manuals, Intel. Available online: http://www.intel.pl/content/www/pl/pl/processors/architectures-software-developer-manuals.html.

[7] Aho A. V., Lahm M. S., Sethi R., Ullman J. D., Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, 2006.

About the author

I am working as a Software Development Engineer in MBB FDD LTE C-Plane's K3 project. In order to facilitate testing of software components we are creating compiler, runtime environment and tools for Testing and Test Control Notation version 3 (TTCN-3) programming language. TTCN-3 is standardized by the ETSI and is aimed to provide well-defined syntax for writing tests. In our daily work we are challenged not only by C++ or Python quirks, but mainly by the tasks related to creation of compiler for over a million-line TTCN-3 codebase.

Michał Bartkowiak
Engineer, Software Development
MBB FDD LTE

U-Boot: How Linux is Loaded on Embedded Systems

Krzysztof Garczyński
Embedded Engineer
MBB System Module

Piotr Rotter
Embedded Engineer
MBB System Module

NOKIA

1. Starting from the PC

Back in the day, **P**ersonal **C**omputer users were often puzzled by the word BIOS showing up during power up. However, few had the courage to dig into what a **B**asic **I**nput and **O**utput **S**ystem actually is and what purpose it serves. As a matter of fact, people often fail to realize that without a BIOS the PC will not start at all.

The primary role of a BIOS is to configure a computer to a point from which the OS (**O**perating **S**ystem) is able to start. The BIOS does not know everything about the PC, neither does the OS. They both rely on each other to do their job. The BIOS covers all the initial basics, and the OS deploys dedicated drivers to cover the details later on. This description merely scratches the surface of computer boot procedures, but it should just be enough to show how important the topic of firmware is.

BIOS originated in the PC realm, which is defined by a family of well known processors sharing a common architecture and a somewhat limited extension capability. Typically a PC consists of a CPU and a set of peripherals spread across the motherboard, interconnected using dedicated high speed buses. Commercials spare the details and simply list a processor, an amount of operating memory, a graphics card model, and the amount of space available on a hard drive.

Although the PC realm boldly advances with each hardware improvement, little has changed in its essence – the list of PC components has stayed largely the same. The realm of embedded systems is quite different and requires a different benchmarking system. For instance there is no such thing as a CPU (**C**entral **P**rocessing **U**nit) or a single processor architecture because an embedded processor is so much more than just a core and its raw power to compute. Most of the peripherals are integrated into a single chip and there is no possibility to replace them. It is even hard to enumerate them without a dedicated map (currently also known as a Device Tree).

The term "embedded" covers an esoteric universe of devices. Thanks to technologies such as **S**ystem **o**n a **C**hip, our world is swarming with numerous processor configurations. Although the set of features they present might seem similar, each of those devices comes with its own distinct configuration. One might say that they all have a sense of uniqueness built into them, much like humans.

By now it should be evident that in terms of comparison, the SoC is closer to a motherboard than it is to a CPU and therefore creating a fixed BIOS for each variant of configuration is not a practical way to go.
The concept to extend firmware along with the hardware seems tempting but its flexibility is often limited by the amount of freedom granted to proprietary solutions. The plain extensibility fails in the case of truly custom systems. Needless to say, embedded is synonymous to custom...

To open the door of an embedded realm one needs an "open" key. The trick is not to allow users to design extensions but for users to extend the extensibility itself. This is where Das U-Boot comes into play.

2. Fancy features found in the silicon

After a few years with MIPS (**M**ultiprocessor without **I**nterlocked **P**iped **S**tages), serving as the heart of Nokia System Modules, a decision was made to replace them with modern ARM based processors. The number of devices running on ARM cores is constantly growing because of their well optimized and documented architecture and scalability. Of course, these are not the only two advantages of ARMs but documentation alone seems to grant interest and support from the open source community.

As opposed to the proprietary BIOS, U-Boot has easily accessible source code and an extension flexibility limited only by hardware capabilities. This degree of freedom is an absolute must because the ecosystem of a core consists of peripheral circuits defined and implemented by manufacturers using their own proprietary methods. Unification and supervision by ARM Ltd. applies only to the core. This obviously leads to bugs and misunderstandings, which cause serious integration problems and frustration. Often the only way to go around hardware problems is through software flexibility. Bottom line is that there is no single, common solution to boot up a new device. This contrasts with PCs hosting a well known, common BIOS and x86 architecture. If you take a closer look at some of the popular ARM development kits (e.g. BeagleBone or Raspberry Pi) you will certainly notice that the common ground starts only after the Linux kernel goes up.

Presumably this is happening because hardware vendors add parts of silicon to adjust products to their share of the market. Sometimes it is just a workaround and sometimes a feature. A good example is the BootROM, found in Texas Instruments devices, which is responsible for setting up processor modes at a very basic level. Usually, it is a selection of storage medium which contains user files such as U-Boot and Linux image. After a power reset every CPU register is set to a default value, so how is it that BootROM can know which medium is connected and has to be selected? For example, by checking which GPIO Pin (**G**eneral **P**urpose **I**nput/**O**utput) is wired to an additional resistor. Then the status of this pin is saved into a configuration register. When we know where to find Bootloader/operating system images, BootROM can take the next step and run an external bootloader: U-Boot.

3. Das U-Boot

U-Boot is a standalone "bare-metal" application, created and maintained by DENX Software Engineering with a great support system consisting of an open source community and device vendors. U-Boot is a very low level piece of software with two phases of execution. The first one is responsible for the detection and initialization of externally connected RAM and using it to store the supporting code (SPL – Second Program Loader). The second phase is dedicated to configuring different peripherals required by the Linux kernel to start e.g. network cards or memories. This stage is the most important because U-Boot provides configuration parameters to the kernel depending on different cases or requests. As it was mentioned before, U-Boot also configures "external" peripherals such as ethernet switches. From Nokia's point of view it is very important because some information from other modules must be processed and prepared before Linux is started. But that is not all. U-Boot implements handling of basic and popular protocols like I2C, SPI, or USB. List of supported drivers includes hard disks, basic API (Application Programming Interface) for file systems, network protocols, and cryptographic API.

Lately U-Boot underwent a reorganization similar to the Linux revolution in the 3.x kernel series. The new DeviceModel was introduced which should standardize the way of implementing drivers and accessing external peripherals. Previously, it was mentioned that ARM CPUs fragmentation is quite painful. This is why the most important directory in U-Boot is arch/ – where all of the source code related to the specific architectures is stored. There we can find implementations for different CPU architectures such as MIPS, PowerPC, or even AVR32. In contrast to Linux, which introduced a Device Tree model to handle the same kernel on different hardware setups, U-Boot still implements the old Linux mechanism of Board files.

The main goal of a Board file is to implement a routine called **board_init_f**. The first lines are target independent and can be found in **crt0.S**.n All it has to do is initialize the stack and memory storage for a GD (**G**lobal **D**ata) structure. But no matter what, GD fields are not available yet. The Next phase is to call **board_init_f**. Depending on the configuration (with or without SPL) execution flow might call other routines like **board_init_r** or **relocate_code**. It is worth mentioning that U-Boot ships with default implementations of these procedures declared using the weak attribute (see **Listing 1**).

Listing 1 board_init_f declaration.

```
void board_init_f(void) __attribute__((weak));
```

The weak attribute marks this symbol as a default one and can be overridden by a developer (during the linking stage the default implementation will be replaced).

The start up and configuration of U-Boot deserves its own book; however, so far the main focus has been on features which are not visible to a typical user.

4. User interaction

One might think that we do not have access to U-Boot because we do not see it. By default, a bootloader needs to stay hidden from mere mortals to avoid panic and confusion. The Android phone users might consider it somewhat disturbing for their screens to show arcane prints instead of a cheery spinning robot logo!

It is quite the opposite for embedded engineers – they love to see what is going on under the hood at all times. Actually if it were not for this exciting curiosity, we would have never enlisted for a job in hardware interfacing. Perhaps this is the reason why we are not scared off by the amount of wiring and sophisticated equipment on our workbenches used to enter the world of embedded devices. This kind of work is what makes us tick!

The break out wiring is key in the early stages of development because it is the practical medium we can use to take a peek at the action going inside a circuit. Usually the first thing which embedded software engineers look for is the serial console. In case of headless systems, which by definition lack a display, it is the serial console which acts as the primary user interface.

Assuming we have set up a serial terminal (C-kermit is your best friend), as soon as the system powers up, we should witness a steady flow of messages coming from the bootloader. At some point it will invite us to play around:

```
=> Hit any key to stop autoboot: 3
```

This is the moment we have been waiting for. With a single keystroke we can enter the U-Boot's command line and start fiddling with all the tools previously enabled during build configuration. The fun does not end here – the final chapter describes a way to extend this command line interface with custom made commands.

The command line is particularly useful for tweaking U-Boot environment variables. These are typically kept in non-volatile memory, making them good for storing persistent system configuration details such as MAC (ethaddr) and IP (ipaddr) addresses, or the invocation of the Linux kernel (bootcmd).

The U-Boot environment uses a very flexible format which allows users to create new environment entries. To create and store a new variable "foo" with the value "bar", all you need to do is:

```
=> setenv foo bar
=> saveenv
```

The saveenv call is required to write the environment back to persistent storage. To "print" the environment entries, all you need to:

```
=> printenv
```

The ultimate goal of using U-Boot is to pass the system configuration through the environment variables to the Linux kernel invocation. This, however, does not limit the possibilities that go with a simple yet flexible environment format. The U-Boot distribution contains an example userspace application called fw_printenv which can be used to access the environment from Linux. U-Boot can pass data not only to the device drivers, the Service Manager, or the operating system's own init scripts – it can pass data to whichever component that might need it!

5. U-Boot environment

Because U-boot came with the Swiss army knife DriversModel for different devices and protocols, it brings us multiple scenarios for booting the Operating System. One of the most popular solutions is a MMC (**M**ulti**m**edia **C**ard) or flash. All we have to do is create and put uImage into the SD card and point U-boot to it. We can do this each time manually but, as we know, engineers are people who want to keep it simple (**KISS** rule) and less susceptible to the reset button. Users who have had a chance to work with development kits are familiar with the uEnv file. This is a so called configuration script for U-Boot. It can be used to define new variables or command sequences to be executed by U-Boot.

On **Listing 2** below, we have a short example of uEnv.txt content from BananaPi development kit.

Listing 2 BananaPi uEnv.txt.

```
aload_script=fatload 0:1 0x43000000 /script.bin;
aload_kernel=fatload 0:1 0x48000000 /uImage;bootm 0x48000000
uenvcmd=run aload_script aload_kernel
bootargs=console=ttyS0,115200 consoleblank=0 console=tty0
disp.screen0_output_mode=1280x720p60 hdmi.audio=1 root=/dev/sdc1
rootfstype=ext4 elevator=deadline rootwait
```

The first two lines of **Listing 2** define where U-Boot will find the binary configuration file for BananaPi periphery and memory address where data will be copied. The same applies to uImage, but as we can see there is an additional step defined in this command, bootm, which will try to boot the operating system from the passed in address. Copying data from SD card or flash to a specific address in the memory is done by the fatload command. This solution is not so flexible as we would expect because the file system type must be known from the beginning. Fortunately development kits with MMC are usually using FAT32 for storing files required by U-Boot.

The next two lines are definitions directly responsible for booting up Linux. The Ultimate command is bootcmd which calls uenvcmd, but only when the latter is defined. Otherwise, a default action will be called and for 99% of the cases it will not fit to the "custom" requirements resulting in a boot procedure failure. In this case, the uenvcmd contains the code to copy data from the card into the memory. This is the default procedure, executed when the waiting counter reaches 0 and there is no interruption from the user. The Bootargs variable is a set of parameters which are well known to Linux users forced to modify their GRUB (**GR**and **U**nified **B**ootloader) on a x86 PC. Linux Kernel does not know where to find its root file system and where the startup scripts are stored. That is why we have to pass all this information from a bootloader. It allows us to prepare multiple booting scenarios depending on the conditions or hardware configuration.

When we reach the first logs from Linux on our screen we can consider U-Boot work to be finished. At this moment everything is in the hands of Linux Kernel developers and we have to trust that their drivers and software will finalize the hardware initialization stage. This trust is mutual because with great power comes great responsibility – the Linux startup is less painful when U-Boot does its part properly.

6. How to start?

Work with U-Boot is not only focused on preparing variables and scripts for Linux. There is still so much to be done regarding new architectures, features, or even code cleanup. The motto "Every engineer is eager to create something new" brings us to the following short introduction on how to add a new command to the U-Boot prompt. Please be aware that we will not describe how to prepare crosscompiler tools and I assume that you, dear reader, know the basics of the C programming language.

Command implementations are kept in files starting with cmd_*.c located in the common/ directory of U-Boot.

What will our first command do? As usual in the first steps, it will print "hello world" to the serial console. Let us start with the code and then proceed to the explanation.

Listing 3 Hello world example.

```
common/cmd_hello_world.c:
#include <common.h>

static int hello_print(struct cmd_tbl_s *tbl, int flags,
                       int argc, char *const argv[])
{
  printf("Hello world in our modified u-boot!\n");
  return CMD_RET_SUCCESS;
}
U_BOOT_CMD(hello_print, 1, 0, hello_print,
"Short help description", "Long help description:
Hello world test");
```

Every command has a specific list of parameters as seen in **Listing 3**. The first one, **tbl**, points to a structure responsible for storing command attributes which are set up by the U_BOOT_CMD macro. The **Flag** variable is only responsible for telling U-Boot in which area this command should be available (BOOTD or environment) or whether it repeats the last command or not. Finally, **argc** and **argv** are argument handlers with the same purpose as it is in normal C programs. As we can see, U-boot also implements basic operations well known from userspace programming as printf (strings operation routines are also available).

But how about the **U_BOOT_CMD** macro? It simply fills up the proper fields in the structure which will be stored in the U-Boot internal commands list. Here, we have to give a unique name (for list only) and name of the command which will be called from the console. There are also two integer parameters: the first tells U-Boot how many maximum parameters to expect (at least one because of the command name) and Boolean describing whether the command can proceed to autorepeat. We certainly do not want an autorepeating command that does flash writes!

As you can see, there is no sorcery here. Of course, advanced features require much more work and the use of different parts of the U-Boot source code. But, it is hard to describe everything in such a short article. We have to dig deeper into it, as we engineers love to do!

About the authors

This piece was contributed by a group of engineers from the Linux From Scratch team. The authors spend their daily time maintaining U-Boot, but when the time comes and a new prototype emerges, they jump out of the routine to balance on a bleeding edge of software and hardware. Living on the silicon edge is what brings joy to their lives and drives them. Their work provides an ecosystem for other teams and lays the foundation for Nokia's most daring technologies.

Krzysztof Garczyński,
Piotr Rotter
Embedded Engineers
MBB System Module

Tuning the Algorithms for Bin Packing Problem

Łukasz Grządko
Senior Engineer, Software Development C++
MBB FDD LTE

Problem statement in Telecommunications Network Planning

Two practical network problems presented in this article are commonly known as Bin Packing problem.

The first issue concerns the licensing in the WCDMA network. Let us consider High Speed Downlink Packet Access (HSDPA) and HSUPA (U for Uplink) schedulers. They have dedicated licenses that are called HSDPA/HSUPA BTS processing sets. Each set allows certain amount of HSDPA/HSUPA users and throughput to be allocated. Each license is paid by the customer who wants to keep costs as low as possible. Since each user requires HSDPA/HSUPA resources, as many users as possible should be allocated to a minimal number of processing sets.

The second issue concerns optimized access network selection in a combined Radio Access Technology (e.g. WLAN, LTE, UMTS, GPRS) environment [6, 7]. Multimode terminals equipped with multiple RAT, are becoming increasingly popular. The procedure of efficient, suitable, and scalable access network selection is gradually becoming an important feature of heterogeneous wireless environments. This is the case for the growing proportion of new handsets being equipped with more than one RAT and wireless access networks of various types. The mobile devices vary in characteristics (e.g. communication range, power consumption) and Quality of Service (QoS) parameters (e.g. bandwidth, delay). It is important to minimize UE power consumption in order to improve QoS and ensure continuous connectivity.

These common problems are computationally hard therefore the computer science focuses on various algorithms that are split to two groups. The first group gives us optimum results whereas the second one cannot guarantee optimal solutions. Unfortunately, all known exact algorithms have exponential time complexity as Bin Packing problems are NP-hard [12]. However, the approximation algorithms very often give acceptable results in a reasonable period of time and therefore, are more practical. In this paper several approaches are shown to find exact and approximated solutions.

1. Problem definition

There is given a set of n items of various size. The goal is to find a minimum number m of bins into which all items can be allocated.

More precisely:
$S = \{w_1, w_2, \ldots, w_n\}$ where $1 \le w_i \le C$ for each $i \le n$ and S is set of items sizes (or weights) w_i each. $\forall_i(w_i \in \mathbb{N})$, $C \in \mathbb{N}$.

Split S to $S_1, S_2, \ldots, S_m$ where $S_i \subseteq S$, for each $i \neq j, S_i \cap S_j = \varnothing$, $\bigcup_{i=1}^{m} S_i = S$, $\sum(w_j \in S_i) \le C$ and m is the minimal possible number. We assume later that S is multiset.

There is also continuous formulation. In this situation we assume $C = 1$ for each bin and $\forall_i(0 < w_i \le 1)$. We can transform the problem instance from natural numbers to rational numbers, dividing each weight and bin capacity by C. Thus if $w_i \le C$ then $\frac{w_i}{C} \le 1$.

1.1. Integer Linear Programming (ILP) formulation

ILP problem is defined as minimization of product value $c^T x$ with constraints $Ax \ge b$, $x \ge 0$. The A is a matrix, c and b are vectors of known integer coefficients. The optimum vector of integer variables x which fits constraints needs to be found. This is computationally hard as ILP is NP-hard. A Bin Packing problem can be transformed into a system of linear inequalities and solved by known ILP methods [2, 19].

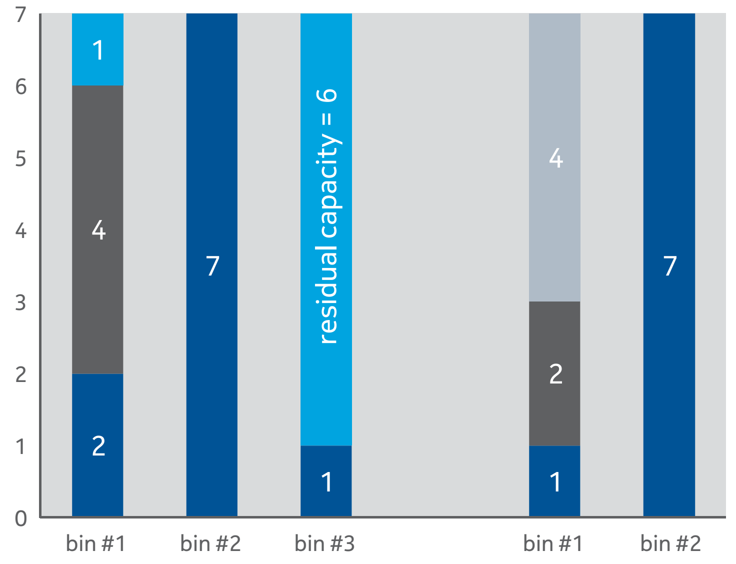
2. Exact algorithms

Few algorithms provide an optimum solution and their computational complexity is exponential in number of items.

2.1. Brute-force approach

First of all, the problem will be solved by determining all possible item sequence. For each sequence items are consecutively allocated to the bin. If the next item does not fit, then it must be allocated to a new bin. Each sequence of items is a permutation. For example, let us consider a set $\{1,2,4,7\}$ and $C = 7$. The feasible solution for this permutation is depicted in **Figure 1**. The number of permutations to check is $\frac{n!}{z_1!z_2!\ldots z_k!}$, where z_i means that i-th unique element of permutation counts z_i times. For multiset $\{1,1,1,1,5,5,5,8,8,9\}$ there are $\frac{10!}{4!3!2!1!}$ (12600) permutations to check. For larger n, there are many possibilities that need to be checked. Let us imagine that a computer checks $10!$ permutations per second. If we apply 60 items there are $\frac{60!}{10!}(\sim 2.29 * 10^{75})$ orderings per second to check. This is equal to $\sim 7.27 * 10^{67}$ years.

Figure 1 Left side shows feasible solution for the permutation {2,4,7,1}. Right side shows optimal solution for the permutation {1,2,4,7} (see also 3.1.).



2.2. Dynamic programming algorithm

Suppose there is only a fixed number r of distinct sizes of items. Let $dp[n+1][n+1]\ldots[n+1]$ be the r dimensional array. A single element $dp[z_1][z_2]\ldots[z_r]$ is the minimum number of bins for packing $\sum_{i=1}^{r} z_i$ items with exactly z_i items of size w_i. Let F be the list of configurations, i.e. a vectors $(k_1, k_2, \ldots, k_r)$ so that the total packing $k_i \leq z_i$ item copies of size w_i does not exceed the bin capacity. During the first step dp is set to 1 for each element from F onwards. During the next step the dp is calculated for a larger number of necessary bins. The recursive formula is as follows:

$$dp[z_1][z_2]\ldots[z_r] = \begin{cases} 0, \textit{ if } (z_1, z_2, \ldots, z_r) = (0, 0, \ldots, 0) \\ 1, \textit{ if } (z_1, z_2, \ldots, z_r) \in F \\ \min\{1 + dp[z_1 - k_1][z_2 - k_2] \ldots [z_r - k_r] : (k_1, k_2, \ldots, k_r) \in F\} \end{cases}$$

The number of different configurations is at most $O(n^r)$. Since r is a fixed number, the complexity is polynomial in n. The time complexity of algorithm is bounded by $(n+1)^{2r}$. The space complexity is $O(n^r)$. In practice, time complexity is bounded to $\prod_{i=1}^{r} z_i^2$. A dynamic programming approach is more efficient for very small r values. Otherwise it is very time consuming. Supposing $n = 200$ and $r = 10$ then the time complexity in a worst case scenario is $(200+1)^{20}$. See in [4] for more details.

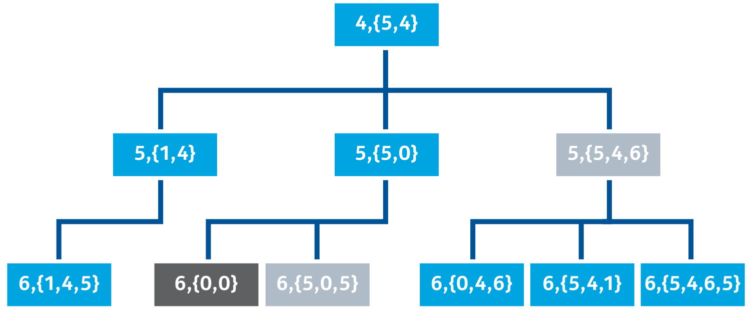
Listing 1

```
int packItem(int index)
{
  int currentResult = INF;
  if (residual.size() == index) //we are in the leaf
    return numberOfBinsUsed(residual);
  for (int i = 0; i < residual.size(); i++)
  {
    if (!visited[index] && items[index] <= residual[i])
    {
      updateResidualAndVisit(i, index);
      currentResult = min(packItem(index + 1), currentResult);
      restoreResidualAndUnvisit(i, index);
      if (residual[i] == capacityOfBin)
        return currentResult;
    }
  }
  return currentResult;
}
```

2.3. Backtracking algorithms

Let us imagine search space as a rooted tree. Internal nodes have a partial solution to the problem instance. Each leaf corresponds to a feasible solution from which at least one leaf contains an optimum result. Each depth level of the searching tree is numbered by an item index. We start with the empty solution, i.e. no bin is initialized and no item is used. The first item is assigned to the root node. At each node, the first free item is consecutively assigned to the feasible initialized bins (by increasing index) and to a new bin. Formally, a node contains $(r_1, r_2, \ldots, r_b)$ vector which denotes the current residual capacity of b bins after allocating the first $i-1$ items. Suppose we try to pack i-th item. Each bin that is initialized in the current node we branch to the descendant node with the following configuration $(r_1, r_2, \ldots, r_j - s_i, \ldots, r_b)$ or branch to node with new b + 1-th bin, i.e. $(r_1, r_2, \ldots, r_b, C - s_i)$. At each leaf node all items are packed and the feasible solution is recognized. Fragment of algorithm implementation is depicted in **Listing 1**. After searching all feasible solutions the best one is chosen. In **Listing 1**, *residual* is a list with a residual capacity for each initialized bin. At each tree level if item can be put to the bin and it has not yet been visited, then the residual capacity is updated and the node in the tree is marked as visited (function *updateResidualAndVisit*). When a leaf is found then we search for another solution. In this case the node is unmarked and the residual capacity is restored (function *restoreResidualAndUnvisit*). There is an example fragment of the searching tree in **Figure 2**.

Figure 2 An example fragment of the searching tree. Each node follows the format *index item*, $\{r_1, r_2, \ldots, r_b\}$. Items weights w_i are: 2, 3, 3, 3, 4, 5, C = 10. At level 4th, two bins are initialized and four items have been allocated.



2.3.1. Branch and bound heuristics

Backtracking methods are based on exhaustive search to which some bound heuristics may be introduced. These heuristics sometimes make it possible to find a solution faster as it is not necessary that all nodes or branches are visited. Before enumerating the candidate branch solution, the branch is checked against upper and lower estimated bounds on the optimal solution [14]. If no ideal solution for the branch can be found then we can leave this sub-problem without solving it.

2.3.1.1. Upper bound heuristic

Recall the **Figure 2**. Note that we do not need to search through nodes marked in gray, as they are a less favorable solution in comparison to the dark gray node, currently best upper bound.

2.3.1.2. Lower bound heuristics, L1 (Christofides and Eilon algorithm)

Temporarily assuming that these items can be broken, each bin (possibly not the last one) can be fully allocated. In this case, the minimum number of required bins is a sum of all weights divided by the bin capacity. Now we can introduce the lower bound formally defined as L_1:

$$L_1 = \left\lceil \sum_{j=1}^{n} \frac{w_j}{C} \right\rceil$$

If a feasible solution is equal to L_1 then we can terminate the algorithm. Recall dark gray node in **Figure 2** for an example. Since the weight sum is 20, then $L_1 = \frac{20}{10} = 2$. Eilon and Christofides [5] present a backtracking algorithm based on the following strategy. Assuming that j bins at any node have been initiated let $(r_1, r_2, \ldots, r_j)$ denote their current residual capacities in increasing order and $r_{j+1} = C$ for uninitialized bin. The branching phase consecutively assigns the free largest item to bins: $s \leq i \leq j+1$, where $s = \min\{h : 1 \leq h \leq j+1, w_j \leq r_h\}$. Lower bound L_1 is used to fathom the nodes. Despite this bound simplicity, L_1 can behave well when experiencing problems with weights that are small in respect to the bin capacity.

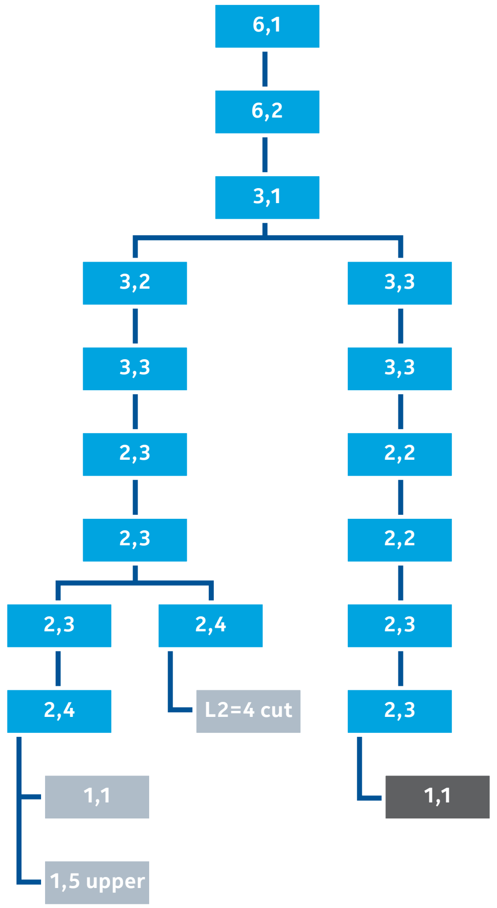
2.3.1.3. Lower bound heuristics, L2 (Martello and Toth algorithm, Korf improvement)

For problems with larger weights, Martello and Toth propose the better bounds, i.e. L_2 and L_3. They are more effective than L_1 bound and solutions are found earlier as longer branch searches are avoided. The idea of L_2 is to add an estimate of the total bin capacity that must be wasted during any solution, before dividing it by the bin capacity. If items are sorted in descending order of their weights, L_2 can be computed in amortized linear time $O(n)$. Notice that at any decision node, some bins are initialized and each bin has some residual capacity. This must be taken into account during computation of L_2. In addition to unallocated items, we add new b items of weights $C - r_b$ for each of b initialized bins. The overall Martello's and Toth's algorithm is based on "first-fit decreasing" strategy where the items are initially sorted in non-ascending weights [5]. Because the Martello calculation is very complex, Korf offers a more intuitive approach [16].

1.3.1.4. Dominance criteria (Martello and Toth)

When the current item i is assigned to the bin b and $r_b \leq (w_i + w_n)$ this assignment dominates all the assignments of items (called dominated) $j > i$ which prevents the insertion one or more further items to bin b. Such assignment closes bin b. There are conditions that cause the bins to reopen [5]. This criteria filters-out the additional branches during the search.

Figure 3 The branch and bound search. Each node has a notation: w, b, i.e. w denotes the item's weight, whereas b denotes bin's number. Items weights are: 6, 6, 3, 3, 3, 2, 2, 2, 2, 1, C = 10.



2.3.1.5. Bin Completion (Korf)

Similarly, bin completion is branch and bound algorithm, but searches different problem space. Rather than considering each item in turn, we consider each bin in turn [16].

3. Approximation algorithms

All previous algorithms have exponential complexities which are impractical for a larger number of items. Therefore it is suggested to use approximation algorithms which have polynomial time complexity. Their accuracy is usually acceptable in practice. We show three greedy on-line algorithms [13], their improvements when input data is off-line and asymptotic approximation schemes. We also introduce the notion of OPT which denotes the optimum solution.

3.1. Next-Fit (NF) algorithm

After allocating the first item to the first bin we then process the next item, followed by checking if it fits into the same bin as the previous item. If it does not then we need to open a new bin. Sample C++ code is depicted in **Listing 2**. Implementation operates in linear time $O(n)$ and constant space $O(1)$. It is proven that NF never uses more than $2OPT$ bins [8].

Listing 2

```
int nextFit()
{//we assume there is at least one item
 //each weight does not exceed bin capacity C
  int nextbin = 1, residual = C;
  for (int i = 0; i < numberOfItems; i++)
  {
    readWeight(w);
    residual -= w;
    if (residual < 0)
    {
      residual = C - w;
      nextbin++;
    }
  }
  return nextbin;
}
```

3.2. First-Fit (FF) algorithm

The difference to NF is that with each step the item that fits is put into the lowest index bin or into the new bin. The straightforward approach uses time $O(n^2)$. The first idea proposed by Martello [5] is to use 2-3 Trees. These trees have dictionary operations (insertion, searching) in time $O(logn)$ and therefore, the whole algorithm has time $O(nlogn)$ complexity. However its implementation is complex [13]. In [1, 3, 17, 18] are simpler data structures described, i.e. Static Trees. Consider the tree in **Figure 4**. The number of leaves should be the smallest power of 2 which is not smaller than number of items. Notice that the tree is sometimes too big if the number of required bins is much smaller than number of used items. The first number in each node denotes sequence number of a node. The second number in each leaf denotes the residual capacity of each bin or, in case of the internal node, the maximum value of its children. The sequence number of node v has a nice property. Their sons have numbers $2v$ and $2v+1$. The parent number is $v/2$. Therefore a tree of this kind can be easily stored in an array. Consider again the tree in **Figure 4**. Suppose we want to add new item with a weight value 9. The first fitting bin has a residual capacity 10 highlighted in gray. When the residual capacity is updated, all nodes should be updated up to the root. It is known that cost of the solution of FF is at most $\frac{17}{10}OPT+1$.

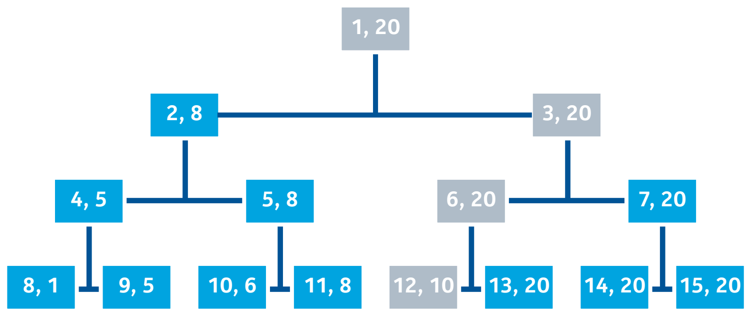


Figure 4 The tree shows its content after inserting the items: 15, 7, 14, 4, 6, 2, 12, 10 in this order. The bin capacity is $C = 20$.

3.3. Best-Fit (BF) algorithm

BF is similar to FF except that it places item j in the bin i whose current residual capacity is the smallest and $w_j \le r_i$. If no such bin exists then a new bin is used. BF can be implemented in $O(nlogn)$ using multiset sucture from std C++ library. A fragment of the implementation process is depicted in **Listing 3**.

Listing 3

```
int bestFit()
{
  std::multiset<int> S;
  for (auto const& item : items)
  {
    auto it = S.lower_bound(item);
    if (it == S.end())
      it = S.insert(C);
    int el = *it;
    el -= item;
    S.erase(it);
    S.emplace(el);
  }
  return s.size();
}
```

3.4. Improvement for NF, FF, BF

When we face offline algorithm, i.e. we are given all the data from the beginning then we may always sort the items in non-increasing order. Afterwards we use pure NF, FF or BF algorithm for such sequences. In **Figure 5** all three algorithms are presented. The cost for FFD is at most $\frac{11}{9}OPT + 3$ [2, 10, 11]. Compare this cost with FF variant.

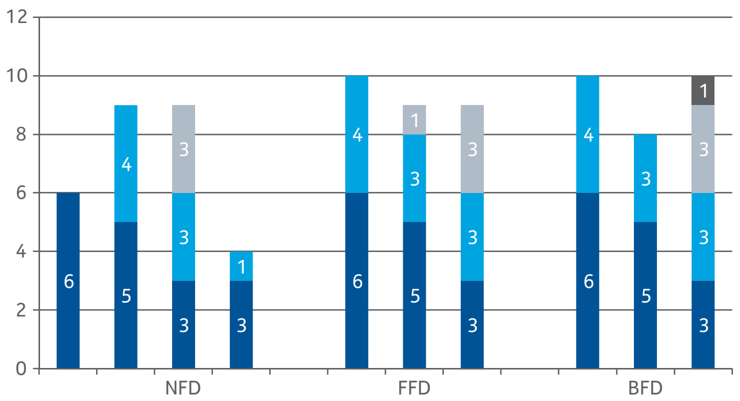


Figure 5 All three algorithms for the same descending sequence, i.e. $\{6,5,4,3,3,3,3,1\}$ and $\mathrm{C} = 10$.

3.5. Asymptotic Polynomial Time Approximation Scheme (A-PTAS)

We sort items in descending order. Items are grouped into large ones, i.e. $w_i > \frac{\varepsilon}{2}$ and small i.e. $w_i \le \frac{\varepsilon}{2}$, where $0 < \varepsilon \le 1$. Larger pieces are split into r groups. In each group all elements are adjusted to largest element in this group. Suppose there are l elements in each group. Then all $l * r$ elements are packed by the dynamic programming algorithm. Afterwards the FF algorithm packs small pieces, possibly opening new bins [2]. The cost of this algorithm is at most $(1+\varepsilon)OPT+1$. Notice that dynamic programming algorithm output is a single number, i.e. the minimal number of bins used. However in this case we need to find a detailed residual capacity for each used bin as well. Using an additional array similar to dp, we might also find out which items are assigned to each bin.

3.6. Karmarkar Karp algorithm for Linear Inequalities

Suppose we have n items and among them are r distinct sizes. Let $F_i = (k_{i1}, \ldots, k_{ir})$ be the $i - th$ configuration (see section 3.1). For each $i \le N$ (number of configurations) we introduce a variable x_i which denotes the number of bins that have F_i. Then integer programming relaxation parameters can be defined as follows:

$$c = [1_1, 1_2, \ldots, 1_r],$$

$$b = [z_1, z_2, \ldots, z_r]^{\mathrm{T}},$$

$$A = \begin{bmatrix} k_{11} & \cdots & k_{N1} \\ \vdots & \ddots & \vdots \\ k_{1r} & \cdots & k_{Nr} \end{bmatrix}, \quad x_j \ge 0.$$

Notice that N is exponential in n and r. Karmarkar and Karp [2] discovered algorithm which solves this LP within an error of at most 1 in polynomial time. The cost of approximation is at most $O(\log(s))$, where s is the sum of all weights of items.

3.7. Generating tests for FF

Several trials show that handy random weights of items do not affect FF algorithm approximation error. An example of this kind of "malicious" test case can be found in [8]. This example can be simplified however with loss of solution accuracy. Constructive proof is shown which forces FF to use $\frac{3}{2}OPT$ bins. Consider items with rational weight that does not exceed 1. Imagine a sequence of $4M$ items of size $\frac{1}{3}+\varepsilon$ followed by $4M$ items of size $\frac{1}{2}+\varepsilon$. The optimal strategy is to pack each pair $\left(\frac{1}{3}+\varepsilon, \frac{1}{2}+\varepsilon\right)$ into one bin. This requires $4M$ bins. However FF allocates small items with a weight of $\frac{1}{3}+\varepsilon$ to $2M$ bins and then large items that weigh $\frac{1}{2}+\varepsilon$ to additionally $4M$ bins. In total FF requires $6M$ bins. Now let's transform rational weights into discrete form. Since $\frac{1}{3}+\varepsilon+\frac{1}{2}+\varepsilon \le 1$ then $\varepsilon \le \frac{1}{12}$. We may assume $\varepsilon = \frac{1}{12}$. Because 12 is divisible by both 2 and 3 then we multiply each weight and bin capacity by 12. As a result we obtain following input data: $C = 12,\ S = \{4+1,\ 4+1, \ldots, 6+1, 6+1, \ldots\}$

3.8. Generating tests for FFD

Generating test scenarios for variants with a descending order is more sophisticated. The details are in [15].

4. Conclusions

Even though searching for an exact solution is a computationally hard task finding a "good" approximation scheme requires more detailed research. For larger number of items we may use parallelized or distributed algorithms. In distributed environments (e.g. grid, cluster, parallel supercomputer, nVidia/AMD GPUs) backtracking methods may be distributed to a number of processing units. In particular we may choose k disjoint subtrees from the searching tree and compute feasible solutions in parallel. Parallel approximation algorithms also exist. For instance FFD can be computed in $\mathrm{O}(\log(\mathrm{n}))$ using $\frac{n}{logn}$ processors. See the details in [9]. In **Table 1** we see the results for seven test scenarios for each of presented algorithms. If there is at least one number in cell, the first one denotes an algorithm solution. If the second number occurs, it is marked in: dark gray for epsilon, blue for number of visited nodes, gray for number of permutations, light gray for number of iterations. N/A – 30 seconds timeout or out of memory occurred on Intel i3 2.5GHz with 1GB memory limit. Notice that the memory usage grows in A-PTAS, for smaller epsilons.

According to conditions and factors (e.g. number of users, amount of radio resources, etc.) we may apply combinations of exact and approximation algorithms. It all depends on the size of the problem.

All the algorithms implementations, test generators and test cases can be found in [20].

Table 1 Results for seven test scenarios for each of presented algorithms.

Number of items/algorithm	30	50	320	30200	random 48	random 70	random 1000000
Exact solution	15	15	96	9006	25	27	-
Brute-force	15 155mln	NA	NA	NA	NA	NA	NA
Backtrack, UpperBound	15 135	NA	NA	NA	25 2mln	NA	NA
Dynamic programming	15 1028	15 866512	96 1bln	NA	NA	NA	NA
Christofides and Eilon, L1	15 135	NA	NA	NA	25 2mln	27 179	NA
Martello and Toth, L2	15 135	15 819	NA	NA	25 82175	27 179	NA
Martello dominance criteria	15 79	15 500	NA	NA	25 2987	27 179	NA
NF	22	22	139	13009	34	35	135911
FF $O(n^2)$	22	16	102	9507	25	28	NA
FF $O(nlogn)$	22	16	102	9507	25	28	124776
BF	22	16	102	9507	25	28	124762
NFD	22	22	139	13009	30	35	134827
FFD	15	19	118	11008	25	27	124635
BFD	15	19	118	11008	25	27	124635
A-PTAS	18 0.38	18 0.5	113 0.88	NA	32 0.38	NA	NA

Table 2 Complexities of each algorithm.

	Type of algorithm	Time complexity	Space complexity
Brute-force	Exact	$O(\exp(n))$	$O(n)$
Dynamic programming	Exact	$O(n^{2r})$	$O(n^r)$
Backtracking algorithms	Exact	$O(\exp(n))$	$O(n^2)$
BF(D)	Approximation	$O(nlogn)$	$O(n)$
FF(D) Brute-Force	Approximation	$O(n^2)$	$O(n)$
FF(D) + StaticTree	Approximation	$O(nlogn)$	$O(n)$
NF	Approximation	$O(n)$	$O(1)$
NFD	Approximation	$O(nlogn)$	$O(n)$
A-PTAS	Approximation	$O(n^{2r})$	$O(n^r)$

References and github resources

[1] J. Radoszewski, „Wykłady z algorytmiki stosowanej", http://was.zaa.mimuw.edu.pl/?q=node/5

[2] D. P. Williamson „Approximation Algorithms", Lecture 2

[3] D.S. Johnson "Near optimal bin packing algorithms" Ph.D. thesis MIT, Cambridge, MA

[4] M.R. Salavatipour, "Approximation Algorithms", Lecture 7.

[5] S. Martello, P. Toth "Knapsack Problems, Algorithms and Computer Implementations"

[6] K. Anderson, Ch. Ahlund "Optimized Access Network Selection in a Combined WLAN/LTE Environment"

[7] B. Xing, N. Venkatasubramaniam "Multi-Constraint Dynamic Access Selection in Always Best Connected Networks"

[8] S. Suri "Course Data Structures and Algorithms, Approximation Algorithms, Bin Packing"

[9] R.J. Anderson, E. W. Mayr, M.K. Warmuth "Parallel approximation algorithms for Bin Packing"

[10] B.S. Baker "A new proof for the First-Fit decreasing Bin Packing problem" Journal of Algorithms 6, (49-70)(1985)

[11] D. Simchi-Levi "New worst case results for the bin packing problem"

[12] J. E. Hopcroft, R. Motwani, J.D. Ullman "Introduction to Automata Theory, Languages, and Computation"

[13] T. Cormen, C. Leiserson, R. Rivest "Introduction to Algorithms"

[14] http://en.wikipedia.org/wiki/Branch_and_bound

[15] G. Dósa "The tight bound of First Fit Decreasing Bin Packing Algorithm Is 11/9OPT + 6/9"

[16] R. E. Korf "A New algorithm for Optimal Bin Packing"

[17] F. D. Alves Brandao "Bin Packing and Related Problems: Pattern based approaches"

[18] https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

[19] http://www.aco.gatech.edu/conference/focs-aco/Karp-lecture.ppt

[20] https://github.com/nokia-wroclaw/nokia-book/tree/master/02/BinPacking

About the author

My computer adventure began at the age of 8 with cousine's C-64 and Amiga 500. Just two years later my own Amstrad CPC 6128 became a wide opened gate for an exciting quest in programming simple games in Basic language. I have participated in many Maths and Programming Competitions and decided to further widen and strengthen my math skills by attending and graduating from Computer Science in University of Wrocław.
I have interests that span the spectrum from theory and mathematics to algorithms and data structures to application and software. Currently I work as a Software Development Engineer in LTE C-Plane. Our engineers create solutions for the new challenges in LTE Mobile Broadband technology using various of software tools and programming languages, mainly C++11/14.

Łukasz Grządko
Senior Engineer, Software Development C++
MBB FDD LTE

Best Engineering Practices

Object-oriented Programming Best Practices

Tomasz Krajewski
Software Developer
MBB System Module

NOKIA

The two most important values of software are correctness and flexibility. The former is achieved with tests (among other things). The latter is important due to rapidly changing requirements. It's achieved with good code structure and high readability. This article is intended to show how to leverage a plethora of useful techniques to make your code not only correct, but more readable and easier to change.

1. Naming

We spend about 80% of development time reading the code [8]. This is why high readability is mandatory. This is achieved by correctly partitioning your code. This will be explained later with proper names and code identifiers. All names should tell you exactly what the code does and use correct parts of speech (see Table 1). Think about other members of your team who will try to understand your intent. Make sure your code is accessible for readers of all backgrounds [1,2].

Table 1 Parts of speech to use when naming.

Identifier type	Part of speech to use	Example
class	noun	ProcessBuilder
method	verb	removeAll()
interface	adjective	Clonable

Example of poor names:
https://github.com/nokia-wroclaw/nokia-book/blob/master/02/OOP_best_practices/MaxPrime.java

Example of good names:
https://github.com/nokia-wroclaw/nokia-book/blob/master/02/OOP_best_practices/LargestPrimeCalc.java

The wider the function scope, the more general and shorter the name should be. You should find that your innermost, specific functions will have long names that leave nothing to the imagination. E.g. in a political poll data processing logic, there could be a function named **`calculateSupport()`**. Deep down the stack it could invoke a function that looks like the following: **`evaluateParliamentMandatesDistributionExcludingUncertainVoters()`**.

2. Structure your code

The well known rule is that each function should have one task. So what does this mean? Well, every time you read your function and you're able to use your IDE's "Extract method" refactoring on a part of your function – you have succeeded! You have just found a subroutine that doesn't really belong to the method as it does something else. So you keep extracting those subroutines. Only when you're unable to extract any other function can you say that your function does one thing [1,2].

If you're dealing with really long methods, you're likely to find that some parts that could easily form a separate class, as they have completely different responsibilities. These functions should be extracted and separated.

So how small can functions get? Ideally 4 lines or less. 6 is ok however, 10 is too much. How easy would it be to read? In four lines you can't fit any if-else, switches, 'try-catches" and 'do-while" clauses. All those statements would be moved to subroutines. This would minimize the amount of indentations, further increasing readability.

Don't worry about getting lost in a vastness of thousands of tiny functions calling other gazillion of even-tinier ones. As long as these sentences are carefully named. Nowadays processors and memory are so fast that you won't notice the delays introduced by calling multiple small functions (unless you're working on a critical part of a real-time application) [1,2].

One more thing you should notice when extracting is that your function arguments should be at a similar level of abstraction. General public functions should operate based on high-level business logic objects. Low-level functions usually deal with low level concepts such as files, arrays of numbers, web sockets etc.

Consider the following code. Its purpose is to accept sentences from console and print basic statistics about those sentences. It's an example of code deprived of any structure:
https://github.com/nokia-wroclaw/nokia-book/blob/master/02/OOP_best_practices/StatsGame.java

Note that various responsibilities are crammed into one method. I/O, processing, formatting, game flow are all mixed up. Any aspect of the application that would like to change is related to the StatsGame module. The module will change a lot and grow very rapidly.

Now consider this refactored code:
https://github.com/nokia-wroclaw/nokia-book/blob/master/02/OOP_best_practices/StatsGameRefactored.java

Changes applied here involve renaming and extracting constants, variables, methods and repeating pieces of code. IO mechanism was extracted to the **`ConsoleIO`** class, as it has nothing to do with the game itself. And interface **`Talkative`** for IO was created. Logic that processes sentences was moved to a separate class **`SentenceProcessor`**. Separate exception classes were created, for readability and easier testability.

Yes, there are more lines of code here. But note that every method is small and easy to read and responsibilities have been separated. Developers can work on modules independently, free of conflicts. Also, with just a little modification, **`ConsoleIO`** and **`SentenceProcessor`** can be injected into the **`SentenceStatsGame`**, which will make it much easier to test. If you aren't familiar with dependency injection, you can read about it in [14].

Using the `Talkative` interface has a fundamental advantage. The underlying IO mechanism implementation is irrelevant to the `SentenceStatsGame.` It means that not only can `ConsoleIO` be easily changed, but also that it can be replaced entirely, for instance by `AndroidWidgetIO`. The `SentenceStatsGame` would be fully operational.

Similarly, if for example a game designer decides that mathematical expressions should also be evaluated then an interface for input processing can be introduced. Next, the sentence processor and the newly created `MathExpressionProcessor` will both implement that interface. After that, adding processors for other types of data will be as easy as adding new `Talkative` implementations which is the idea behind the process. Flexibility, allowing programs to adapt to customer needs [1,2].

3. Form

How many arguments should a function have? The more arguments there are, the more complex the function gets, and the more ways there are for it to behave. Therefore it's best that your functions have no arguments. Consecutive function calls would then read like a prose, wouldn't they? Of course you have to pass some arguments every now and then, but fewer is better. Three arguments is the (barely) acceptable maximum [1,2].

But what kind of arguments are acceptable? Any kind is certainly doable. But think about booleans. Consider such a method:
https://github.com/nokia-wroclaw/nokia-book/blob/master/02/OOP_best_practices/FilesReporter.java

Would you extract anything from the `listFiles` function? Of course, the code inside the `if` statement and the code inside the `else` statement. So the `listFiles` function really consists of two functions. Therefore it is best to delete this function and create `listFilesNonRecursive(String folderPath)` and `listFilesRecursive(String folderPath)`. Actually in the end these two methods would go into two separate classes that implement an interface, but that's a topic for another article.

So, using boolean arguments this way is a huge mistake. Usually the same goes for enumerators, which indicate forthcoming complications. Such enumerators almost always can be changed into polymorphic constructs.

The whole team should agree to use the same coding convention. When a team member reads the code, they should be unable to tell who wrote it based on the style used. That helps the code to be predictable. For example, you could agree to use the same code formatter. This kind of formatter dictates indentations, number of empty lines, maximum line width, etc.

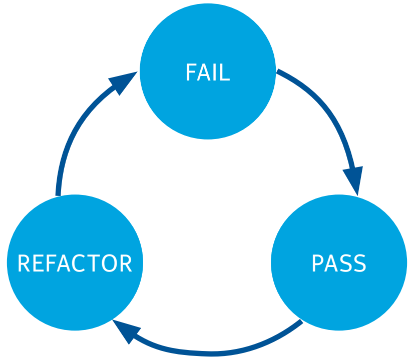
Are comments a good solution? One the one hand comments can provide vital information however, on the other instead of explaining what part of the code does, you should just give meaningful name to that part. Additionally, the comment easily becomes obsolete when the code being commented changes. Therefore comments are not desired. The code itself should indicate what it does. If you need a comment to explain the code, then the code needs to be rewritten [1,2].

4. TDD basics

Test Driven Development is a process that consists of three consecutive steps repeated over and over again (see also Figure 1):

1. Write a unit test that fails (compile error is also considered as failed).
2. Write code that passes the test. But write as little code as it's sufficient for the test to pass.
3. Refactor your code if necessary.

Figure 1 The flow of TDD.



TDD is a technique that – if applied correctly – is guaranteed to cover 100% of your code and 100% of use cases.

Sounds impossible? It can, at first. But consider this. Do you like writing unit tests? Most developers I know don't. Have you covered all cases with a set of well written unit tests that you spent a long time preparing? Do you feel protected by those tests? Would these tests detect bugs that you might introduce to your code?

What almost always happens is that even a large suite of carefully written tests doesn't protect you very well from errors. You write tests when you're finished with implementation, so you already know that the code works. It is impossible to think about all the test case scenarios. Not to mention that it's dull.

This is not possible in TDD. Also, you will never write code that isn't testable, especially code that produces side effects such as the method `add(a, b)` that not only returns a sum of `a` and `b`, but also prints something to the standard output or sends the result over the network to a remote host. The TDD regime alone won't allow it.

But the key benefit is that tests written with TDD offer the ability to enable developers to edit existing code and refractor it freely, as the tests will detect if the code is damaged.

Suppose there is a method `divide(double m, double n)` we want to test. It should return `m/n`. For the sake of example let's omit issues related to floating point precision. In TDD, we start with simplest cases, easiest to handle:

1. Thinking about possible values of n, we come up with a test `errorWhenSecondArgumentIsZero()`. The test will use the method `divide` and will expect that `15.0/0.0` will throw an exception. The test will fail. It won't even be compiled.
2. Create a method `divide(double m, double n)` which throws the exception that the test wanted. That's the simplest thing possible that makes the test successful. Make sure is passes the test.
3. The refactor step – nothing to do yet.
4. Write a test `numberDividedByOneReturnsSameNumber()`. The test will expect that `15.0/1.0` = `15.0`. The test will fail because the `divide` method throws an exception.
5. In the `divide` method add an `if` statement before the throw clause. `if n==1.0`, `return m`. That's enough for the tests to pass.
6. Still there's nothing to refactor yet.
7. Write a test `divideNonTrivialNumbersReturnsCorrectResult()`. Use the `divide` method a few times with a different set of operands and appropriate results.
8. After that `if` statement add `else if n!=0.0` clause. In `else` block, there's nothing we can do to make the tests pass except for writing the actual code, that returns `m/n`. The tests should now pass.
9. Finally we get to refactor. Observe that the `if` statement can be removed and `else if` can be changed to `if`.

Surely this is a trivial example but even here we got our code coverage and all the use cases are also covered.

5. Cohesion and coupling

Cohesion is a feature of a class. It indicates a degree to which functions of the class have single, well-focused purpose.
Examples of highly cohesive classes:

- `Calculator`, with the following methods: `add()`, `substract()`, `multiply()`, `divide()`.
- `ProgressBar`, with the following methods: `reset()`, `incrementByOne()`, `incrementPeriodically(long milisecs)`, `stopIncrementingPeriodically()`, `setToDone()`.

Cohesion pays off because such classes are easier to reuse across various modules. Also, these type of classes are less frequently changed, as they're focused around just one purpose.

An example of a poorly cohesive class: A class that adds a user to a database. Its methods: `createUser(String email, String password)`, `saveUserToDatabase()`, `printTransactionResult()`. Three functions that have very different purposes: database operations, creating database objects and reporting operation results.

Here is one observation that could be helpful when determining cohesion compliance. It's how functions use fields. If every function uses all fields, it's very likely that we're dealing with high cohesion. But if, say, three functions use a set of fields that the other two functions do not use then perhaps those two kinds of functions are not focused on the same purpose. There are metrics for cohesion (like LCOM4 for Java) although the notion itself is subjective.

Coupling is a degree to which one class knows about another class. If changes in one class affect how another class works, those classes are highly coupled. That is undesired as it makes it hard to change single modules, because other modules that depend on it will break.

Consider the following code, a part of College application.
https://github.com/nokia-wroclaw/nokia-book/blob/master/02/OOP_best_practices/College.java

If we decide to change a table to `List` in the `Group` class, the `StudentsPerGroupStats` won't compile. Coupling takes its toll. In this case, we should add the `getStudentsCount()` method to the `Group` class. That way the data structure in `Group` will not depend on the `StudentsPerGroupStats` class.

The refactored code:
https://github.com/nokia-wroclaw/nokia-book/blob/master/02/OOP_best_practices/CollegeRefactored.java

6. DRY, KISS, YAGNI

DRY stands for Don't Repeat Yourself. Whenever you hit that Ctrl+V in the code, you may regret it later. It's a signal that maybe some module can be extracted and then reused. If you paste, you create more code to maintain and to test. Remember: "Less is more".

KISS means Keep It Simple, Stupid. As Albert Einstein said, everything should be as simple as possible, but not simpler. Think about your readers. Quite often tough problems remain tough for everyone until they're solved by someone who didn't know it was tough. So it's possible that one of these days a rookie will come to your team and will implement the same algorithm in a much simpler way. And you don't want that.

YAGNI is short for You Aren't Gonna Need It. Plain and simple; don't write code that is not required right now. Don't litter your application by code that nobody needs.

7. Law of Demeter

This is also known as Don’t Talk To Strangers. This is how methods rely on members of other classes. If a class **`A`** uses field **`f`** from a class **`B`** or invokes a method **`m()`** from a class **`C`**, it’s all good. We look at the class **`A`** and we see right away it depends on classes **`B`** and **`C`**, so field **`f`** and method **`m()`** are “close” in terms of how many modules we have to go through to get what we need.

On the other hand, if a class **`A`** reaches out for a field **`f2`** from class **`B`**, to invoke **`f2`**’s method **`m7()`**, just to get the third object from a collection that **`m7()`** returned and invoke **`toString()`** method on it… Note how complicated it gets. Think about how tightly all modules in that long chain of invocations are coupled. And how hard will class **`A`** be to test [9].

There is no easy solution to when code violates the Law of Demeter. In order to avoid this refactor your code, keep the cohesion high, move code around through the modules until you find one that has the same purpose, encapsulate your variables, and utilize dependency inversion [1,2]. All these actions lead to classes that are loosely coupled, which usually makes Law of Demeter easier to follow.

8. Handy tips

Here are some hints coming from experience.

1. **Reinventing the wheel** is harmful. When coding, ask yourself how likely is it that someone else already had to write code that does that very same thing. If it’s possible, search both within your project codebase and on the web. There might just be a class for that. There might be a library for that. Examples for popular Java libraries: Guava, Apache Utils. The best code is the one you didn’t have to write. Less is more.
2. **Code coverage** is useful, but only to indicate if you have forgotten to test a specific part of the application. Getting 100% coverage does not necessarily indicate that you have finished. Just analyze your test suite to make sure that all the viable use cases are tested. Note that you can get 100% coverage and not make a single assertion, therefore testing nothing! That said, not all the code is worth testing, e.g. getters and setters.
3. **Util or Helper classes** usually grow into a cluster of methods that are unrelated. Developers looking for a place to add a new utility method are likely to put it into this kind of bag. As the cluster grows it becomes a point that many modules depend on. Instead of feeding utils, try to find a class that your method really belongs to and put it there. Don’t forget to make it private, too.
4. **Comment heavy wizardry.** [13] As said in the chapter about form, comments are wrong and this is generally the case except when your code depends on third party modules that you’re forced to use and are either unusual or buggy. In this case you can write an explanatory comment, e.g. **`“//workaround due to this library bug 7352`**”. Or “**`//no way to extract actual measurement data with this API, so the whole plain text measurement log is parsed to find it`**”.
5. **Scoundrel method signatures.** Suppose there is a web browser that has a module to debug websites. This module could have the method **`addBreakpoint(file, lineNo)`**, but the method has an unusual side effect. If your breakpoints have been deactivated previously, the **`addBreakpoint`** will in fact activate them! Another common example is methods that don’t use their arguments. Like **`displayWindow(topLeftVertex, size)`** that for some reason ignore the given size and determines the actual size based on the content in the window. Don’t ever allow method signatures become deceiving.
6. **Broken windows theory.** It’s a criminology theory that describes relation between the state of urban environment and the rate of crimes. Preventing small crimes like vandalism fosters the sense of order. That in turn makes people less prone to committing serious crimes. This theory also applies to software. If you keep your code in order, others will be reluctant to corrupt it with some poor code. On the other hand no-one will care about code that was poorly written in the first place.

9. Professionalism

Most projects in software development suffer from gradual yet severe decrease in development velocity as the project matures. It has often been the case that adding new code to a project proved too difficult causing managers to decide to start the project from scratch, only to experience the same problems a few years later. This is caused by not caring about keeping the code clean. Despite our obligations for timely delivery of our products, we do not see this as our top priority. Poorly coded software has caused many tragedies such as; car break malfunctions; space shuttle explosions; patients being subject to fatal radiation levels during therapy. Therefore it is our responsibility to our customers to only provide software that is fit for purpose. It’s a sign of a professional, to tell people who has hired them what to do, not the other way around. Your insight should determine how code should work and if it’s ready to be delivered, not the deadlines that others impose [3,11,12].

10. Summary

In this article we’ve only scratched the surface of creating clean code. There are many more rules and good practices such as SOLID principles [1,2], agile development and design patterns. Unfortunately most of these can only be fully understood whilst coding.

No rule is sacred though. Sometimes it’s a good idea to break a particular rule, but only when there is a good reason for it. E.g., Even if there is a good reason to break the rules you do need to understand inside-out in order to bend the rules without causing repercussions [5].

References

[1] Martin R., Clean Code: A Handbook of Agile Software Craftsmanship, 2009
[2] http://cleancoders.com/category/clean-code#videos
[3] https://www.youtube.com/watch?v=p0O1VVqRSK0
[4] Feathers M. – Working Effectively with legacy code, 2004
[5] https://vimeo.com/43536417
[6] http://www.planetgeek.ch/wp-content/uploads/2013/06/Clean-Code-V2.2.pdf
[7] http://www.slideshare.net/guest446c0/the-smells-of-bad-design
[8] http://blog.codinghorror.com/when-understanding-means-rewriting/
[9] http://haacked.com/archive/2009/07/14/law-of-demeter-dot-counting.aspx/
[10] https://sourcemaking.com/refactoring/bad-smells-in-code
[11] http://www.devtopics.com/20-famous-software-disasters/
[12] http://en.wikipedia.org/wiki/Sudden_unintended_acceleration
[13] http://en.wikipedia.org/wiki/Deep_magic
[14] http://www.vogella.com/tutorials/DependencyInjection/article.html

About the author

I’m a Software Developer in MBB SM SCM and Automation. We provide tools, expertise and assistance that help other teams within Nokia to develop and test our products. One of our tools is an RCP-based environment to write automated tests using a script language we designed for that purpose. Another example is a tool for analyzing network traffic that extracts desired data based on a set of parsers and filters. We care about quality of our software and it pays off with the ability to implement new features quickly, even for mature products.

Tomasz Krajewski
Software Developer
MBB System Module

Best Engineering Practices

SOLID Principle – Finding the Root

Dawid Bedła
C++ Software Developer
MBB FDD LTE

1. SOLID short description

SOLID is one of the most popular acronyms in the world of Object Oriented Programming. Popularized by Robert C. Martin in the early 2000's, the acronym stands for Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion principles. These rules should be well-known by programmers who want to write high quality code.

1.1. Single responsibility principle

The single responsibility principle describes class and method as elements which have a dedicated task. Therefore we assume that class has only one reason to change itself. Class focuses on successfully completing a specific task. This makes the interface easy to use and understand. In other words, all classes and interfaces should be short while the method should contain only a few lines and all used names are as verbose as it is possible. This makes code easier to read and maintain in the future.

1.2. Open-closed principle

The open-closed principle dictates software to be closed for modification and opened for extensions. Following this principle, the class should not change defined behaviors. Adding a new feature is easy and has almost no impact on already implemented code. It is achievable by properly distributing functionalities. This can be done by putting functionalities in dynamic libraries with well-designed interfaces. Single libraries can be easily updated when the function's behavior needs to be changed or extended without recompiling the whole program.

1.3. Liskov substitution principle

Liskov's substitution principle is a fundamental rule.

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.[2]

This sentence describes how polymorphism works. Polymorphism is the method of changing the behavior of our code by injection of a new class that implements the interface in a different way.

1.4. Interface segregation principle

The interface segregation principle indicates to whom the interface belongs to. Previously the interface was attributed to the class that implemented the interface, now it is the class that uses the interface as gateway to the external code. The module defines the way of communication with themselves. This means that the module knows what and how the way of communication can be changed e.g. which part of functionality can be injected (data source, data presentation layer, detailed algorithm implementation) and what is core functionality which should stay unchanged.

1.5. Dependency inversion principle

The dependency inversion principle dictates that code becomes independent from concrete class implementation but instead dependant on interfaces. It makes code easy to reuse in other projects or libraries. It is possible when external code cooperates with library/framework via well defined and documented interfaces.

2. The root

Polymorphism is *The Root* of all SOLID principles. A well designed polymorphic interface makes it possible to write code that is easy to extend and maintain.

Virtual functions are language constructions that implement polymorphism, this conception is available *for free* in some object oriented languages (C#, C++). From a programmer's point of view, it only requires an extra keyword during definition. Polymorphism is so important and such a basic functionality for object oriented programming languages that some of them (Python, Ruby, Objective-C, Java) make all functions behave like virtual functions. There are two main ways to implement polymorphism: virtual table of pointers to functions and sending message to object.

3. Good abstraction

Now for an example that shows how good abstraction makes code flexible. Consider the following problem: "Write a module which read, process and present data". This is quite a standard problem. It is easy to extract three basic functionality by reading data, processing data, presenting data. On closer inspection, we can also find out that core functionality is processing data. Reading and presenting are the parts of the code that will be probably modified in the future. While writing the module it will be useful to mock or inject own implementation of these operations.

Now when problem analyses is finished it is time to make some concrete assumption about input data, the way of presentation and processing. Let's say that data is a simple collection of integers which can be read in grouped portions. Processing is the operation where for every single input number we get one recalculated output number. Presenting the data procedure present data to the user or making it available to further processing. At this point the programmer has a lot of information about what the application should look like. Unfortunately there are no details necessary to write specific code. There is no information on how data is provided: from file, external device like sensor, read from network or user provide data from keyboard. The processing description is still missing calculation detail (math equation or other transformation parameter). Polymorphism allows the code presented on **Listing 1**.

Listing 1

```
struct DataSource {
  virtual std::vector<int> readData() = 0;
  virtual ~DataSource() = 0;
};

struct DataPresenter {
  virtual void presentData(const std::vector<int>&) = 0;
  virtual ~DataPresenter() = default;
};

struct DataModifier {
  virtual std::vector<int> recalculateData(
  const std::vector<int>&) = 0;
  virtual ~DataModifier() = default;
};

struct Application {
Application(DataSource& ds,
          DataPresenter& dp,
          DataModifier& dm,
          std::function<bool()>stopMethod) :
            dataSource(ds),
            dataPresenter(dp),
            dataModifier(dm),
            stop(std::move(stopMethod))
  {
  }

  void runApp() {
    while(!stop()) {
      auto data = dataSource.readData();
      auto dataAfterModifications =
        dataModifier.recalculateData(data);
      dataPresenter.presentData(dataAfterModifications);
    }
  }

private:
  DataSource& dataSource;
  DataPresenter& dataPresenter;
  DataModifier& dataModifier;
  std::function<bool()> stop;
};
```

The presented code allows to separate application logic from implementation details. Information about concrete implementation of *DataSource*, *DataPresenter* and *DataModifier* has to be provided only in one place. For this application, it will be a main function where instance of *Application* class is created.

Listing 1 shows how important the proper usage of polymorphism is. In this case business logic is simple and can be kept in one place (*Applicatnion::runApp*). In more general situations, business logic is much more complicated and distributed through many classes. Interfaces make separate work on business logic code and implementation code possible(which also can be extremely complicated). This approach is very convenient during all stages of software lifetime.

4. Polymorphism in various languages

This paragraph describes how different languages implement polymorphism.

4.1. Duck typing

It is time to discover language implementations details. At the beginning we will see why *duck typing* allows programmers to use all functions as virtual functions. The description is based on Objective-C implementation (it is syntactical most interesting language) but it is very similar to languages like Ruby or Python. Consider code presented on **Listing 2**.

Listing 2

```
[instance method:argument];
```

This is basic construction for calling the *method* by class object *instance*. The name *instance* represents an instance of a specific class, *method* represents the method name for class and *argument* represents the parameter passed to the function call. Objective-C is a language with named parameters. The function calls are unusual due to them having multiple parameters. See some *real example* on **Listing 3**.

Listing 3

```
//declaration
- (void)drawCircleAtPoint:(CGPoint)p
              withRadius:(CGFloat)radius
               inContext:(CGContextRef)context;

//call
[instance drawCircleAtPoint:midPoint
                 withRadius:size
                  inContext:context];
```

Character "-" in front of method name indicates a standard class method ("+" indicates a static class method). By looking at the declaration and function call it is possible to split it by ":". Everything on the left side of ":" is the function name, on the right side is an argument.

Such construction forces the programmer to create more readable and well-named interfaces. This is very important in big projects with long time support.

Knowing the basics, it is possible to focus on how the function call is conducted in message based languages. All classes derive from base type (NSObject). Base type provides meta-data about user-defined classes including method name lists. The name of the method must be unique. After compiling the code from **Listing 3** we get following classic C function call.

Listing 4

```
objc_msgSend(instance, drawCircleAtPoint, midPoint, size, context)
```

Function *objc_msgSend* checks if the given object has implement *drawCircleAtPoint*. If yes, the function will be called by its given parameters. If the function was not implemented *nil* value will be returned (Python and Ruby throws an exception). *Duck typing* approach consumes more memory and runtime performance. Language with a communication based on messages does not need a strict inheritance hierarchy. It might look very tempting but it has some flaws.

4.1.1. Duck Typing – dark corner

The biggest problem for dynamic languages is the unintentional override function of parent classes and modification of base class behavior. The programmer can override the parent classes by writing their own function with the same name and parameter list in his own class as method in base class. There is no easy way to avoid this. For Python or Ruby, the programmer has to manually check if the function has a unique name in the inheritance hierarchy. It is even harder for Objective-C where private methods are not reflected in class interface.

4.2. V-Table

The second option is to use a table of pointers to functions called virtual tables or v-tables. This solution was chosen for C++, C# and Java as it is faster but requires a strict inheritance hierarchy.

C++ has different mechanisms for virtual and non-virtual functions. For non-virtual functions compiler checks type of object/pointer/reference and uses the correct one. For the class which has at least one virtual function, instances contain special member called v-table (virtual table). This member is *invisible* for the programmer while writing the code but it is possible to check the presence and the value during the debugging phase. This special member contains a list of pointers to virtual functions of a current object. All calls of virtual functions are done via v-table, so even when using a pointer or reference to base class the address of the function is taken based on runtime instead of compilation time data. A v-table is not directly accessible for programmers however it is still possible to change its values. Very often this is the source of serious problems (**Listing 5**).

Listing 5

```
struct A {
  virtual void f() {
    std::cout<< "A::f()\n";
  }

  virtual ~A(){}
  int a;
};

struct B : A {
  void f() override{
    std::cout<< "B::f()\n";
  }

  int b;
};

auto b = std::make_unique<B>();
std::memset(b.get(), 0, sizeof(B));

std::cout << b->a << std::endl;
b->f();
```

On first inspection everything seems to be in order. There are two classes *A* and *B*. *B* is a subclass of *A*. *A* has a defined virtual method *f()* and class *B* overrides this function. Class *A* contains *a* member and class *B* since inherited from *A* contains *a* and *b*. Both classes do not contain a constructor. So instead of writing a constructor which initializes members it is possible to use a function typical for operations such as *memset*. It works for class members. Unfortunately a *v-table* is also a member so *memset* sets all fields to 0. As a result of the program calls virtual function the pointer is attributed with the value *null*. This causes the program to crash.

5. Interfaces in design patterns

Almost all design patterns are based on hiding implementation details behind one common interface. This proves that polymorphism is the most important feature of Object Oriented Programming Languages.

Design patterns implementation can differ from language to language. A good example of this Factory. The pattern's base concept is to create various objects without exposing creation details. A benefit of using this pattern is that all users will be forced to rely on a common interface instead of a concrete class implementation. There are many ways to implement this pattern in C++ one of the best was presented in [1].

Listing 6

```
template <typename T>
class Factory {
public:
  static Factory<T>& getInstance() {
    static Factory<T> instance = new Factory<T>();
    return *instance;
  }

  T* createObject(std::string className) {
    return creationList[className]();
  }

  bool addObjectToCreationList(std::string className,
                  std::function<T*(void)> creationFunction) {
    if (isObjectOnCreationList(className)) {
      creationList[className] = creationFunction;
      return true;
    }
    return false;
  }

private:
  bool isObjectOnCreationList(const std::string& className) {
    return (creationList.count(className) > 0);
  }

  std::map<std::string, std::function<T*(void)>> creationList;
};
```

This is a smart template factory implemented as a singleton. Construction in method *getInstance* is thread safe for compilers which support the C++11 memory model. The most interesting thing is the registration method. Standard *Shape* class hierarchy will be used to present how it is done. *Shape* interface is presented on **Listing 7**.

Listing 7

```
class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};
```

As a registration example we use class *Rect* which is based on the *Shape* interface. Please focus on the *createRect()* method. The *Rect* class registration should be placed in *.cpp file of this class.

Listing 8

```
class Rect : public Shape {
public:
  virtual ~Rect(){};
  virtual void draw()
  {//implementation is not important
  }

  //static method used as creation method in registration
  process
  static Shape* createRect(){ return new Rect(); }
};


//Rect.cpp
namespace {
  const bool addRect = Factory<Shape>::getInstance()
                       .addObjectToCreationList( "Rect",
                       Rect::createRect );
}
```

All *const* variables have to be initialized at the beginning of the application lifetime (in this case even before reach main function). *Rect* class will be available in *Factory* from the beginning of runtime.

All classes which use *Rect* do not have to know anything about it beside that Rect derive from Shape interface. Shape client code can be completely independent from concrete implementation. Using string as the key in factory for a specific object makes it possible to deploy new shape-derived classes without recompiling components that use *Factory* as source of shapes. It wouldn't be possible when using constructions like *enums*.

Factory design pattern is a good example of degeneration based on language polymorphism implementation. This is clear after comparing C++ implementation from **Listing 8** with *implementation* in more dynamic languages.

Listing 9

```
#python
newInstance = getattr(modul, class_name)()

#ruby
instance = Object.const_get('ClassName')

#ruby with rails
newInstance = "ClassName".constantize.new

//objective-c
id newInstance = [[NSClassFromString(@"ClassName") alloc] init];

//C#
var newInstance = System.Reflection.Assembly
                  .GetExecutingAssembly()
                  .CreateInstance("ClassName");
```

Looking at the example in **Listing 9**, it is easy to conclude that in Python, Ruby, Objective-C and C# a factory pattern is available as part of language. This is even more the case for dynamic languages as there is no need to use interfaces directly. *Duck Typing* is the way to avoid information about type of object.

Observer Pattern[4] is the concept where one class *Subject* which maintains data allow other objects *Observers* to register for changes. It is done via defined interface. Class *Subject* provides a method for registration and deregistration; *Observers* provide method for notification.

For static typed languages like C++ it is easy to verify if *Observers* implement the proper method. Registered objects just have to inherit from interface (abstract class) and then implement it(otherwise there will be compilation error). *Duck Typing* allows inheritance to be avoided and the base object to be used for registration instead. This may lead to the utilization of the wrong implementation method during class registration. To avoid runtime errors, verify if the specific method is implemented for the given object instance. It is done by the function provided by the standard language implementation.

Listing 10

```
//objective-c
bool isMethodImplemented = [instance respondsToSelector:@selec-
tor(methodName:)];

#Python
if getattr(instance, 'methodName', None) is not None:
  isMethodImplemented = true

#Ruby
isMethodImplemented = instance.methods.include? 'methodName'
```

6. Summary

Polymorphism is the most essential part of Object Oriented Languages. Different languages have different approach to implement this functionality. It means that some concepts which are valid for static typed languages, do not apply to dynamic deduction types. The implementation of polymorphism impacts specific implementation of design patterns and changes many code writing rules. It is easy to verify this sentence by comparing the source code of two projects: one written in C++/C#/Java second one implemented in Python or Ruby.

References

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Professional, 2001.

[2] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* Pearson, 2002.

[3] Marc Gregoire, Nicholas A. Solter, Scott J. Kleper. *Agile Software Development, Principles, Patterns, and Practices.* Wrox, 2011.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

About the author

I work as programmer in Nokia in Poland. Throughout my career I have used various technologies and languages : C#, Objective-C, C, C++, Python, CUDA. Currently, I work in MBB FDD LTE *Where programmer not only write code but also create software.*

Dawid Bedła
C++ Software Developer
MBB FDD LTE

A Brief Introduction to the Software Configuration Management

Andrzej Lipiński
R&D Manager
MBB System Module

Software development in large scale projects is a very complex process. Large scale – meaning hundreds or thousands SW engineers working in parallel to achieve the project goal. The rule is simple – the more complex the project becomes, the more people are involved. "More people" means more interactions, dependencies and communication. uch large scale introduces complexity which needs to be addressed appropriately. Two or three software developers working on relatively simple web page will effectively coordinate their work and solve the approaching problems on the fly. A different picture will be seen in project working on operation system or in projects we realize in NOKIA like e.g. software running on the radio base stations (Flexi) or the software running the radio network management system (NetAct). The people working in teams made of dozen and more people may spot quite challenging situations. It is sometimes not easy to get a clear vision of the impact of own changes in the source code on the other team members work. Then, even more challenging becomes to realize and understand the impact we made on projects we do cooperate with.

Figure 1 More people means more interactions, dependencies and communication. Cooperation within small team looks trivial comparing to what we face in complex project.

If we want to secure the realization of the software project goals and make it successful, it becomes necessary to introduce rules which would make putting together the software as automated and effortless process as it can be.

It becomes necessary to set clear rules helping various teams working together and cooperate smoothly, e.g. agreeing on "human interfaces" translating one team's needs into other team's activities, agreeing on the roles and associated responsibilities. Agile methodologies come here in handy supporting projects with list of best cooperation practices and making daily work more efficient. Additionally the cooperation in big projects spans not only across different teams but different countries, time zones and cultures as well. Those are the factors the cooperation rules need to take into consideration and utilize them to the project advantage.

It is also necessary to equip the teams with tools which will quickly put together their separate contribution into one working application.

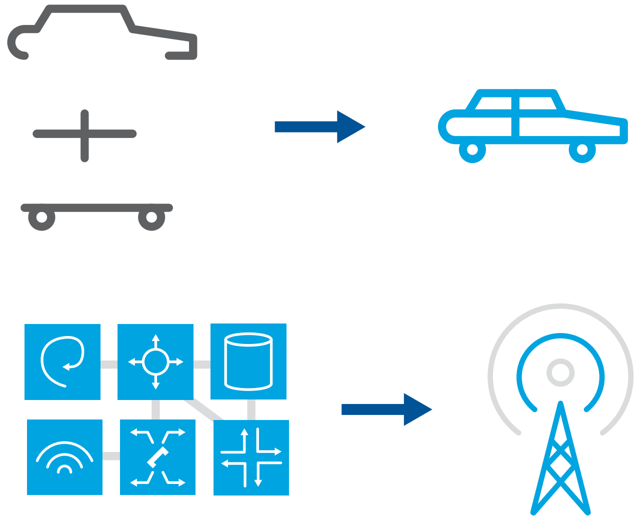
It is exactly the Software Configuration Management (SCM) duty to equip software development projects with appropriate processes and tools which would allow effective creation of subsequent software releases. The other words – the main task of SCM is protecting projects from becoming lost within several versions of software developed at the same time and throughout whole life of the project.

1. Some analogies

The applications, similarly as e.g. cars, are built of the various components cooperating with each other. Such components are created by teams highly specialized in their own domain or in cross-functional teams with team members of distinct expertise areas. The outcome of work of such teams (creating e.g. reversing sensor, transmission layout, etc) makes an input for another team putting all the parts and pieces together into one functional car. On the typical movies from car factories we can usually see whole rows of robots arms working 24/7 on the production line. They are continuously putting together new models of cars from the available components.

Such production line has its counterpart in the software production world and it is called "software integration". Dedicated systems are busy with completing sets of the right components of the application (right – means here in appropriate version, containing appropriate functionalities and verified by appropriate quality tests) and putting them together into one functional application.

Figure 2 Putting all the parts and pieces together into one functional equipment. No matter what industry – the challenge remains similar.



In both cases – car assembly and software production we need rules and appropriate tools which in the most automated manner will gather outcome of various teams working in parallel on different components and put them together in correct way. Those rules describing the way we're cooperating on the daily basis are called "processes".

3. Cooperation

As mentioned, two scenarios are possible here. We can work in "component teams" focusing on next versions of the components of the application. The teams consist in such scenario from highly specialized members and with the deep knowledge about specific component of the application. Scope of their work is to implement new functionalities within the frame of the component. In our car industry example it would be e.g. teams working on the air-condition system or rear mirror design. After their work is being completed new car models are being assembled with latest features coming from those teams.

We can also choose to create "feature teams" – focusing more on whole application functionality rather than single component. The approach follows here the definition of certain application functionality which implementation requires tight alignment of the features in several components. Cross-functional teams are consisting from experts bringing knowledge about different components and working together to develop the functionality coupling features from several components. That would make sense for the car makers e.g. when rain detector experts would join their forces with the wipers system engineers or in order to deliver auto-pilot feature experts from transmission layout, engine, navigation and ambient conditions tracking system would create one team.

In the various software projects we would see both of such team set-ups. The small project would start typically their adventure with cross-functional team – consisting from developers, testers and devops. The "devops" role may be described as part time developer – part time IT/SCM expert; both actively working in development field and – because of the expertise – being able to set-up the working environment for whole project.

Next, when the project grows, it usually makes sense to develop highly specialized teams focusing on their expertise areas. This is the moment when dedicated SCM teams are being created. Their aim is to serve whole project with unified and most efficient tools and processes forming best possible software production line.

However depending on the project needs we can also spot those cross-functional "feature teams" in the big scale projects as well. The SCM teams support such cross-functional development teams in the same way – delivering the means of putting whole project together into one functional application.

4. Configurations

In order to make easier moving around the software projects it is good to agree on certain way how to name all the elements of the work environment as well as the relations between them.

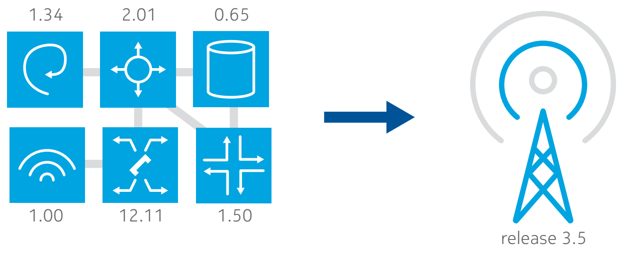
Usage of the "Version Control Systems" (VCS) becomes common practice among software development projects. The main aim of VCSs is to keep safe and organized the subsequent "versions" (or alternatively "revisions") of our code developed across many "branches".

Successful "build" (without compilation errors, without linking issues,..), successfully verified by various "test collections" (showing no functional problems) can be marked as one of the "versions" of our application and can be "delivered" to our customers.

All information related to above elements and the relations between them allowing us to uniquely identify particular version of our application are called "configurations".

Configurations tell us what components were included into specific configuration and in which version, what tools were used to build the components (and in which version) and what tests (and in which versions) were used to verify them. Completing such information helps us to recreate at any point in time the exact working environment and analyze the raised problems.

Figure 3 All information allowing us to uniquely identify particular version of our application is called configuration.



The inconvenient truth is the software projects are not able to discover on the daily basis all the errors existing in their source code. It is project specific how high it sets the bar and how close its products will come to the 100% error free zone. In order to assure that appropriate quality applications are thoroughly tested for extended period of time before they reach the market. The quality of tests and amount of the test scenarios is an object of constant improvement. In order to make bug solving fast, it is necessary to be as much meticulous when gathering the configurations. Thanks to fully automated work environments, we are able to re-create any particular environment (get the right version of the source code from the Version Control System, get the appropriate version of the compiler, system libraries, tests) and instantly start the work on bug fixing.

Very similar, when the project will decide to start work on alternative version of the application – once the decision about which version of the current application will make the foundation for the new project, the work can start after few commands setting up the new working environment. It is possible because the configurations work like recipes – helping us quickly set an appropriate working environment.

5. Coordination

The second mentioned aspect of SCM – cooperation rules – grants us the knowledge necessary to identify who is who in our project and what are the responsibilities in our project family, with whom we ought to talk, whom to listen to, who is obliged to pay attention. As in every family such knowledge of the rules can save us some bloodshed.

Those rules (or "processes") are well known by us and instinctively followed on the daily basis and become practically invisible. They become clearly visible once they are not followed.

Let me get back to the example of the car assembly line. The team responsible for transmission layout has made a major improvement (reliability increase) and replaced all the cabling from copper to fiber optic. They actually also did not bother to communicate this change to anybody outside the team. The team starts to deliver new versions of the transmission layout only to discover major issues raised by reversing sensor team and some other as well. Everybody is still expecting that information about transmission layer dealing will arrive on their copper-based interfaces. The improvement becomes a bug.

The missing part here was the lack of adherence to the cooperation rules described by project processes – which main and overriding task is to secure completion of working car. In above example we may see typical reasons for projects to get into trouble – both lack of sticking to the project coordination rules (teams cannot introduce autonomic not agreed changes to the project in their components) and lack of appropriate communication (the improvement attempt should be widely communicated to the rest of the project in advance). The equivalent of such situation in software development project would be introduction of new functionality to the source code without any consideration of impact on the rest of the project beforehand. The components stop matching each other, application stops working, project enters failure mode.

The cooperation rules aim to describe the safest way for the project to introduce changes.

6. Control

Each complex software development project is based on numerous teams cooperation. Team members must coordinate the new ideas created every day and following implementation so the project could thrive without obstacles generated by mutual interaction of changes introduced. The cooperation between team members must result in timely delivery of new software versions.

It is necessary to make as clear as possible for everybody, what are the interactions between project components and what are the dependencies between them. At any given time during daily work we must exactly know what part of the application we are currently working on. Only then we can easily focus on improvements we're currently struggling with, knowing our changes will be introduced in correct version of the application.

Controlling both the changes of the source code (bug fixes, new features) and application configuration (versions of the compilers, system libraries, ..) in all releases of our project assures we will be able to recreate any variation of our application. It means we must track not only changes introduced to the source code – we must know what version introduced any particular functionality or fixed some bug, but also track all the information about all the tools we did use during writing the code. This makes possible e.g. to identify the problems generated by usage of different system libraries by the team members or spot the problems generated by newer version of compilers introduced to the project.

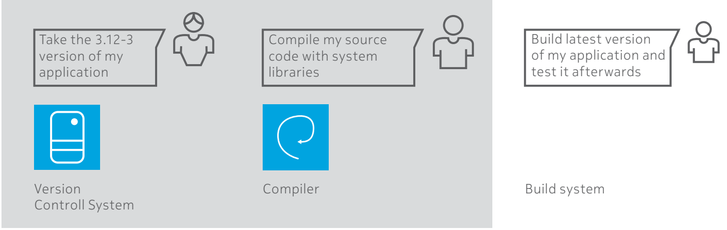
7. Tools

The basic tools used in software development projects are mentioned already: Version Control Systems (GIT, Subversion, Perforce, Bazaar, Mercurial, ClearCase – to name a few), compilers, build systems and continuous integration systems. Additionally common practice is to use in house tools created by the project members themselves – usually using scripting languages like bash, Python, Perl.

Version Control Systems (VCS) are meant to track subsequent versions of the source code. Each time when the programmer becomes happy with the current state of his work he saves it to the disk making it persist. Well, this is not enough – we cannot let the previous version to be overwritten and lost forever. This is the place where version control becomes handy. After saving the content of the file we do inform VCS to record this newest version. One command – and we're done. The new version of our file joins the previous versions in the VCS database and remains safe as long as the database exist. The usage of such tools becomes natural with the first day and becomes quickly a habit. At once we do receive safe way of keeping the history of development of our project and easy way to find any changes we introduced in the past.

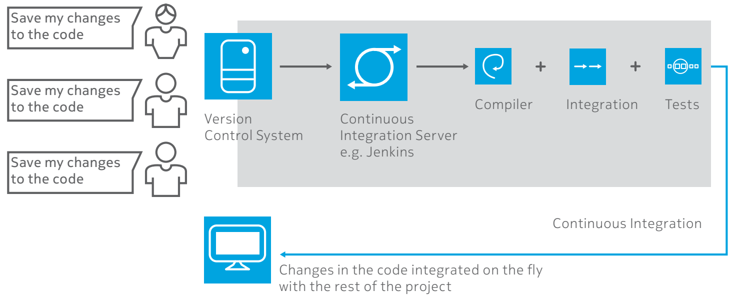
The compilers and build tools (make, ant, maven) are usually built into the "build systems" shaped according to specific project needs. The main task of build systems is to prepare working environment for the software developer, get the right source code version, assure all related application components availability and build a functional/running version of the application we're developing. The build system automatically does all the steps necessary and you can choose if it should build the whole application or only the specific part of it.

Figure 4 The build systems prepare working environment, get the right version of the source code, assure all related application components are available and build a running version of the application.


Take the 3.12-3 version of my application
Compile my source code with system libraries
Build latest version of my application and test it afterwards
Version Controll System
Compiler
Build system

The continuous integration systems (a good example would be Jenkins) use build systems, VCSs and make use of defined sets of tests collections in order to deliver in the real time information about project progress. It is especially useful when a lot of changes in the source code are being introduced by different programmers at the same time. Continuous Integration makes possible for all team members to track the impact of own changes on the whole project. Especially it allows to quickly realizing if the changes do not break the application.

Figure 5 The continuous integration systems use build systems, VCSs and sets of tests to deliver the project progress information in the real time.


Save my changes to the code
Save my changes to the code
Save my changes to the code
Version Control System
Continuous Integration Server e.g. Jenkins
Compiler
+
Integration
+
Tests
Continuous Integration
Changes in the code integrated on the fly with the rest of the project

8. Summary

You can think about all the rules – naming of the elements of our project, rules to control their changes (configurations) – as names of the streets, cities or districts. They are meant to help us better realize what particular place in the project are we dealing with at the moment.

Setting the rules describing cooperation on software development projects helps us to create safe work environment. So even the best programmers – when developing this next groundbreaking approach or smashing this vicious bug – would focus only on what matters instead of worrying about the impact of the rest of the project on their work. They do not need to worry where the specific version of the source code they were working on last week is to be found. They use the project working environment and simply go with the fun work – implementing new features. Unlike the colleague next to them wasting time on investigating issues caused by the latest – just downloaded from internet – compiler.

The processes and tools delivered by Software Configuration Management are meant to remove such problems. They aim to make everybody's life simpler and help work become fun again.

The presence of the rules creates quite interesting side-effect to our job. By default the rules create restrictions for the daily work – not everything is allowed. Sometimes in the heat of the fight with the especially pesky bug, bending the rules seems like the only option to the developers. To our surprise sometimes such approach delivers impressive outcome. Such rule-makers vs rule-benders race is another of means of keeping the progress evolve and makes our work interesting. This constant need for better ways to create the software without compromising the safety of the whole project remain our main challenge.

About the author

I'm working in the MBB System Module SCM department. We take care about everything what is needed in order to make work on WCDMA, LTE or 5G the most efficient. Our portfolio includes build systems, software integration, automating the development work environment – be it with GIT, SVN, Jenkins, Python or bash – we make it work. We make the environments for really large scale projects – where thousands of programmers work each day on the newest generations of the latest telecommunication solutions. We work in teams. We like to look for holes in the whole and make the good solutions better ones.

Andrzej Lipiński
R&D Manager
MBB System Module

Advanced Branches Utilization in Subversion and Git

Marcin Gudajczyk
Senior Engineer, Software Configuration
MBB System Module

1. Introduction to branches in Version Control Systems

Version Control Systems (VCS) have proven to be very powerful tools. Since their appearance in the 1970s, they have evolved significantly over time and have constantly offered more advanced code management possibilities. When given to the developers and integrators, they can be compared to the finest armor, shield, and most of all a weapon which can be used to shape and slice the code itself. Besides basic code storage possibility, VCS provides a wide range of functionalities that make life for people working with code much easier. One such useful feature is branching. To explain branching, picture yourself looking at a tree starting from the trunk at the bottom and moving up to the top step by step. You will notice its branches and even branches of branches. Logically, the VCS structure looks quite similar, starting from the first commit (root) and tracking the history of changes. There are “nodes” where branches are appearing and starting a new development line. They can be used for managing dedicated features made on the top of specific code revisions, or testing various changes. Modifications made on such paths are always available and never lost (unless intentionally deleted). However, unlike in the real tree example, branches can be consolidated to take their specific changes back to the trunk (and not only).

Imagine a really big project. A BIG project in NOKIA reality means that there can be up to 300 developers working continuously on one repository (where code from dozens of repositories is used to build one product). What a mess there would be if they all worked on one “main development line” (the trunk). A single developer trying to implement a feature could face notorious code changes from other people while updating the code with modifications made by others. Those could introduce conflicts or even bugs. To avoid such situations a programmer can simply branch-off from the trunk and work on its own “copy”. While working on such a branch, progress will not be interrupted and the process of putting completed implementation back to the trunk can be handled later. Continuous Integration (CI) is intensively using branches for wide purposes of constant building, testing, and delivering reports of changes made for the project. This article will describe shortly advanced branches utilization procedures with CI usage examples.

2. Representation of commits in Subversion and Git

To further present how both VCSs are managing branches, the method of keeping information for each **commit** (single chunk of changes put into repository with its unique ID) has to first be explained. Both Subversion and Git systems provide data to the user in the form of a directories/files tree in which code can be kept. However, implementation of the method of how each of them stores data differs drastically. This actually determines a few things such as how fast the repository operations are performed, how much the workspace weights, how the user can refer to data, and most importantly what functionalities can be given by VCS.

Subversion documentation claims, that SVN can be described as a simple file system with revision control [1, 2]. Each revision stores only the differences (**diff**) to the previous revision, for example revision no.123 contains the information regarding what exactly has changed comparing to revision no.122. (there are also additional properties available to be defined for revisions, but this is not important in this article at the moment; please check [3] for more information). This actually leads to the fact, that the Subversion repository is a “database” of differences used to prepare the workspace. Appropriate difference reports can be generated for two custom revisions [4].

On the other hand, there is Git which offers an alternative approach. Revisions are not held as differences to the previous versions of repository. Instead each revision is a snapshot of a data. Whenever Git notices, that a file has changed, it stores it to **snapshot** (entirely). If the file was not changed, only a pointer to the previous revision for that file will roll to the snapshot [5]. This creates a repository where data is held in a manner of a Directed Acyclic Graph (DAG), where snapshots are nodes and pointers are directed lines connecting nodes [6]. Novice Git users often have problems in understanding the dissimilarity between a directory tree and a history graph tree. There is also a possibility of generating differences between given revisions.

3. Branching in Subversion

3.1. Basics

As already mentioned, SVN behaves just like a file system, therefore the creation of new branches is handled by a copy operation of a directory (branch cannot be created from a file, neither in Git). This is done by a command supported with commentary in quotes:

```
svn copy svn://repo/source_dir svn://repo/branch_dir \
-m "branch created from source directory"
```

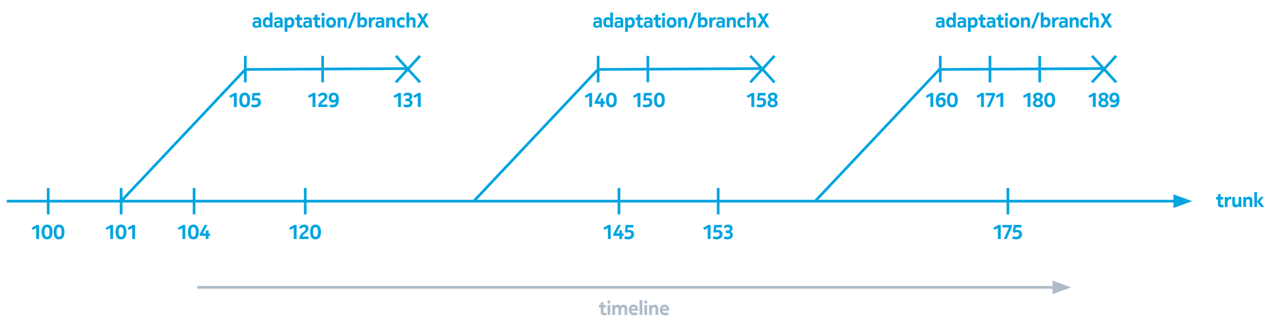
Since SVN is a centralized VCS, each branch is always visible for all repository users. As regular directories, these can also be moved, renamed or even deleted:

```
svn move svn://repo/branch svn://repo/new/location/branch \
-m "branch location has changed"
svn delete svn://repo/new/location/branch \
-m "branch removed from the repository"
```

But what happens with the branch after it is deleted? Was not the repository meant to store the revisions history? Actually it does. If for example a “branch” was deleted in commit 145, it is still visible in all the previous versions where it existed. However, to recover such information a special reference by peg revision has to be used [7] unlike regular revision query “**-r**”, which allows to examine the repository at the state which it was in the past:

```
svn info svn://repo/new/location/branch@144
```

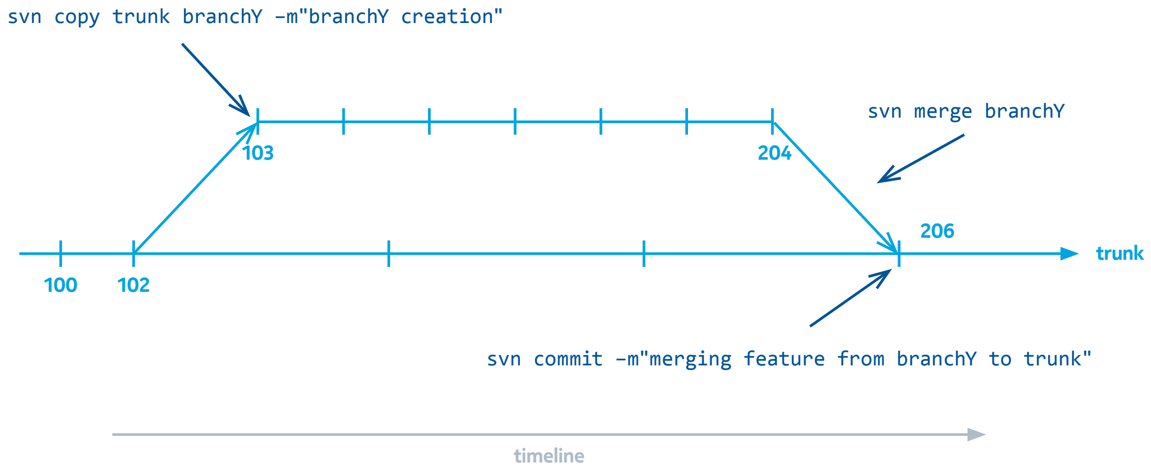
Figure 1 Adaptation branch recreated under the same location. Vertical lines represent commit numbers and crosses are commits responsible for branches deletion. Commits no. 105, 140, and 160 represent "`svn copy svn://repo/trunk svn://repo/adaptation/branchX`". Commits no. 131, 158, and 189 are a visualization of "`svn delete svn://repo/adaptation/branchX`".



NOTE: All arrows on figures related to Subversion represent a timeline and flow of actions made to the repository. Commits (many are omitted to simplify images) will be shown as vertical lines.

CI tip: This solution is extremely useful for handling short living branches dedicated for adaptation. Imagine a mainline with constant environment changes, which breaks all building and testing. A developer's work is ruined every day by such updates. To bypass this problem, integration should be performed on dynamically created branches. After it is finished, a stable code should be put to trunk. This guarantees a stable and working code base for all programmers.

Figure 2 Representation of a merge operation in SVN. Commit no.103 is point of branch creation. Commit no.206 is the merge of changes from branchY to trunk.


```
svn copy trunk branchY -m"branchY creation"
svn merge branchY
svn commit -m"merging feature from branchY to trunk"
```

On the latest revision of repository, the deleted directory does not exist, so a new one can be created in its place. Such an approach allows having distinct branches with identical names in the same location. Each will still be available after deletion, if referred to by the proper peg revision. All basic branching operations are visible in **Figure 1**.

3.2. Merging

Let's assume that a feature was developed on a branch ("branchY" on **Figure 2**). Nearly 50 commits were made to finish it and now it has to be put to trunk. There is no need to perform all the same commits once again onto trunk. The fastest way of delivering changes is to perform a **merge** operation, which takes changes from the source branch and creates only one commit to the target branch, containing all required changes. An important additional attribute is created during this operation – a property called svn:mergeinfo which contains information about the merge source. It is used by SVN to track the change history [8]. It is required to perform such an operation on the local workspace after checking out the repository. Whenever any **conflict** is met it has to be resolved [9]. The merge operation is performed as follows on the target directory (trunk):

```
svn checkout svn://repo/trunk
cd trunk
svn merge svn://repo/branchY .
# Possible conflicts will be visible while performing above
# operation
svn commit -m "merging feature from branchY to trunk"
```

3.3. Cherry picking

There are cases, where specific changes (not all) made on one branch should be copied to another. For this purpose a special merge technique was created called "**cherry picking**". It allows literally picking a specific range of commits or just one commit and introducing the required modification to the destination branch. For the example below, the trunk directory is already a current workspace and a particular commit change from branchY is being picked:

```
svn merge -c 110 svn://repo/branchY .
```

If the merge was extensively used on the repository in the past, it is highly probable that additional data totally unrelated to the commit will also be caught in the operation. Instead of picking changes for requested commit, after executing above command, unrelated modifications will also be introduced. To avoid such a situation, flag "**`--ignore-ancestry`**" has to be manifested for merge operations [11].

3.4. Reintegrated merging

Feature branches are often long lasting creations and have to be maintained for a period of time. Trunk is constantly growing so it is wise to update the branch and resolve any misalignments step by step in order to avoid big discrepancies between the two. For this purpose, regular merge operations are performed through the lifecycle of the branch, however it will pick the changes from trunk to branch (opposite as in the previous example). Such an action can be performed multiple times until the feature will be completed. The final merge will be performed in a regular manner from branch to trunk but with one additional flag "**`--reintegrate`**" (**Figure 3**). This will literally tell Subversion that the branch was synchronized with its parent and special tracking of changes has to be used [12]. The following are some example actions:

```
svn copy svn://repo/trunk svn://repo/branchF -m "branchF is ON"
svn checkout svn://repo/branchF
cd branchF
# Do some changes and commit to the branchF
# Changes were also done on trunk by other users
# so synchronization should take place
svn merge svn://repo/trunk .
# Solve all possible conflicts and commit the merge
# Do some changes and commit to the branchF
# perform synchronization again and commit it
svn merge svn://repo/trunk .
# It is time to put the changes on trunk
svn checkout svn://repo/trunk
cd trunk
svn merge svn://repo/branchF --reintegrate .
svn commit -m "final merge of branchF to trunk"
```

4. Branching in Git

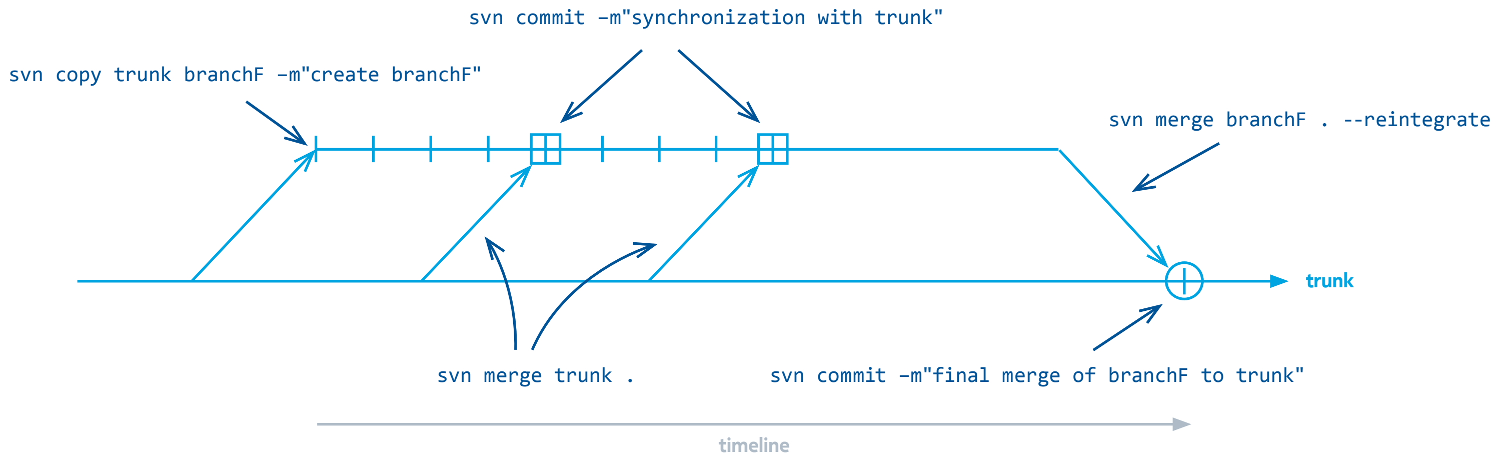
4.1. Basics

From a technical point of view creating a branch in this VCS is a totally different operation. Beside the fact that the commands used are unlike in SVN [13], something else is also done underneath. While a branch is raised, an additional pointer appears to indicate a specific commit on top of which subsequent revisions will be put. The pointer will always move to the newest commit of this branch [14]. In this implementation, the branch gets its full meaning. In a decent repository, graph structure representation will definitely depict the presence of real branches. To create such branches, assuming that the repository is already **cloned** [15, 16], execute:

```
git branch myBranch
git checkout myBranch
# Above commands can be substituted by one below
git checkout -b myBranch
```

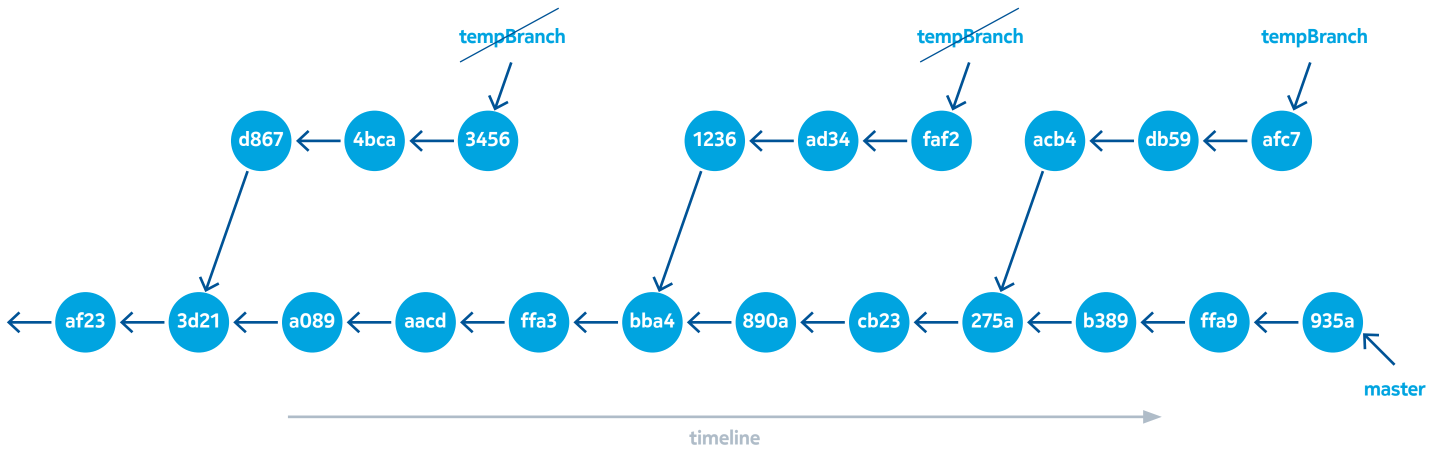
The current data snapshot presented in the workspace is used by default as the base for a new branch. Therefore, the aforementioned commands have only "myBranch" given as the target that will be created.

Figure 3 Visual representation of the maintenance process of a long lasting branch with **reintegrated merge**.


svn commit -m"synchronization with trunk"
svn copy trunk branchF -m"create branchF"
svn merge branchF . --reintegrate
trunk
svn merge trunk .
svn commit -m"final merge of branchF to trunk"
timeline

CI tip: In Subversion 1.8+ the "**--reintegrate**" flag does not have to be explicitly defined and SVN can recognize by itself when to use it; however, for sanity reasons and code clarity there is still profit in keeping this flag visible in all scripts.

Figure 4 Simplified Git repository graph with **master** (SVN's trunk equivalent) and dynamically recreated branch with identical "tempBranch" name. Strikethrough pointers represent operation of a branch "deletion". Commits with 2 pointers are nodes from which a branch was created.



NOTE: All arrows on figures related to Git will represent pointers but not the timeline! Commits/snapshots are shown as circles with abbreviated hashes.

CI tip: Short living branches can be handled this way. CI configuration will not have to be changed because a branch will always be visible under one name.

Keep in mind that unlike SVN, Git is a distributed VCS. Local copy of **origin** is self prosper and fully functional repository. One of the consequences/profits of this is that newly created in example "myBranch" is local. It will not be visible to any remote repository (origin) user until **pushed** [15, 16]. To do this, the following command is required:

```
git push -u origin myBranch
```

In conclusion, the branch name in Git is just a pointer to a specific commit. Once this fact is acknowledged, understanding Git mechanics comes with much ease. So what is exactly happening when a branch is "deleted" (**Figure 4**)? Of course only a pointer will be removed! All commits made for that branch will still be available in the repository while referred by its **hashes** [17] (revisions identifiers represented by SHA-1 checksums, which are often abbreviated, for example "a1d3"). This again leads to the situation where a branch with the same name can be created multiple times in the repository. This should not be done unless a reasonable CI purpose exists.

4.2. Merging

Basic **merging** also differs slightly from what was already presented for Subversion. The operation creates an additional commit, which instead of keeping diff with additional metadata, holds two pointers to the commits that are being merged. Files that conflict during the operation, after resolving it, will be put into the snapshot. This also moves one of the pointers depending on which branch the merge is being performed [18] (please check "master" pointer move on **Figure 5**). Simple commands are responsible for this operation:

```
# Current branch is master
git checkout -b myBranch
# Do some changes here and commit them
git commit -m "changes made on myBranch"
git checkout master
# Update repository content with pull command – other users
# made changes on master
git pull
git merge myBranch
```

The merge operation creates responding snapshot by itself, so no additional "commit" is required here anymore. If no conflicts [18] are met or there are no changes on master, fast-forward [19,20] is performed for relocating/creating pointers. Merged branch can be deleted later on as the changes are now also visible on the mainline. This is however not recommended – "deleting" a branch would only make history less transparent.

4.3. Merging transparency

Because of pointers usage no additional information similar to svn:mergeinfo is required. This allows the merge operation to be performed in both ways (**Figure 6**), from master to branch and from branch to master, without defining any additional flags similar to "**--reintegrate**". Simplification of operations brings more clarity to all actions made on the repository. There is no need to remember about any obligatory flags.

4.4. Cherry picking

This operation is also available for Git and again, because there are no possible issues related to additional metadata, it always picks specific changes from snapshot without any hidden inheritances. It can either transport a single change or a range of changes (one by one) to the required branch. While this operation is performed, totally independent commit(s) are created that have no relation to the snapshot from which it was picked. Here are examples of performing such operations while already working on the destination branch (workspace is the default target):

```
git cherry-pick a12f
git cherry-pick df9a^..cb5e
```

4.5. Rebasing

The last operation worth mentioning in this article is called rebasing [21]. It allows the branch "base" – a commit to which the first branch snapshot points – to be changed. All branch specific commits are taken one by one and applied step by step on a defined snapshot. In comparison to merge, which creates an additional consolidating commit, rebase involves moving existing branch commits (**Figure 7**). Despite that the branch name will not change, commits hashes will! Of course conflicts can be visible here also. Unfortunately, this operation is not desired for published changes of shared branches. It can break other users repository clones while changes are pulled in the default way. It is highly discouraged to alter the history of already pushed commits. Such modifications can also be easily overwritten by unaware users. Rebase is performed by executing:

```
git checkout -b myBranch
# Do some changes and commit them
git checkout master
git pull # Some changes made by other users appeared on master
git checkout myBranch
git rebase master
```

5. Conclusions

From the perspective of this article a statement can be made, that both VCSs provide functionally similar solutions; however, they are implemented with totally different approaches. There are very few tasks that could not be completed in one of the discussed systems. To have a better understanding, history visualization and its variations can be studied [22, 23]. There are also useful tools that provide graphical interfaces for history browsing like TortoiseSVN, TortoiseGit and gitk.

Figure 5 Example of a simple merge operation in Git, where master's pointer is relocated. Strikethrough pointer was active before the merge operation.

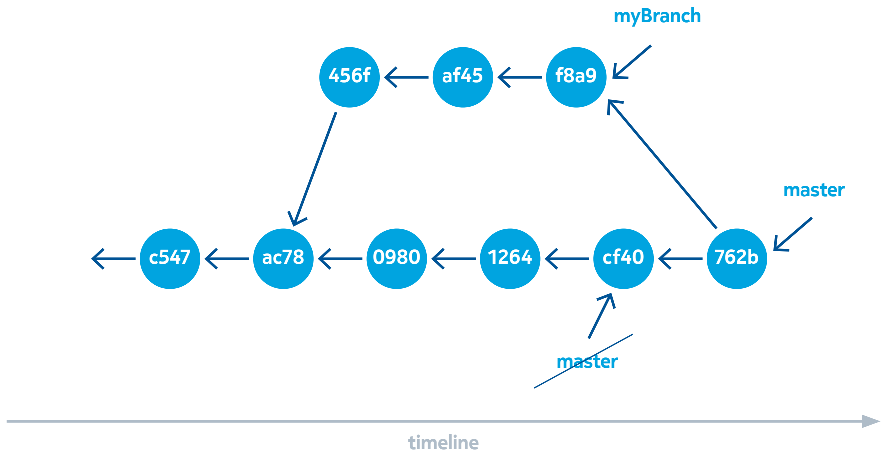
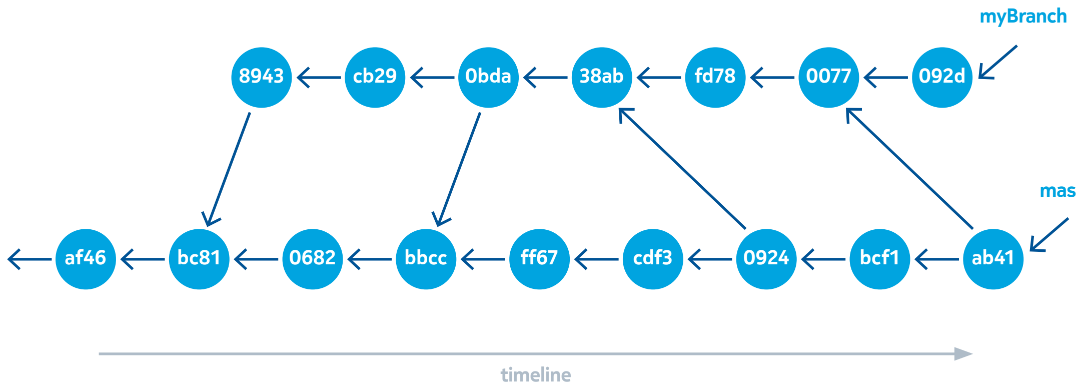


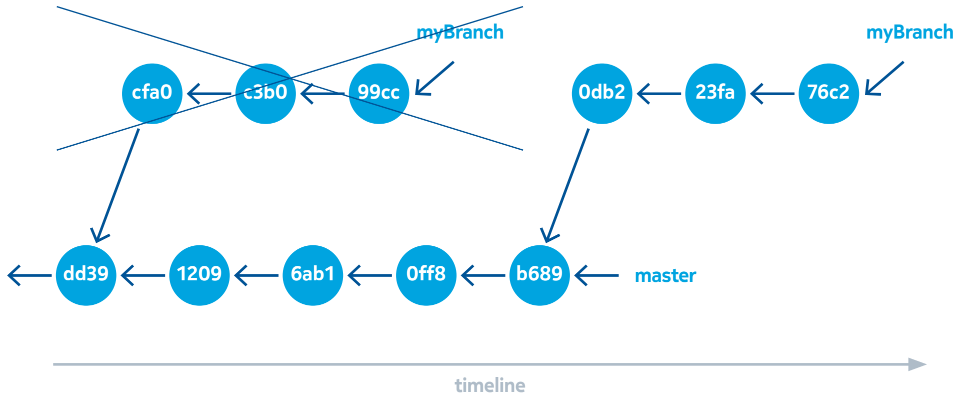
Figure 6 Representation of a multidirectional merge for a branch and its parent. There are many merges from master to "myBranch" and from "myBranch" to master.



CI tip: Git merge implementation greatly simplifies maintenance of long-living branches, because of its universal usage. In comparison to SVN this also reduces amount of code used for scripts handling actions done on branch.

Subversion is much easier to master because it has logic similar to regular file system. Its popularity lies in its simplicity as a tool, however more sophisticated setups are often hard and time consuming to implement. On the other hand, Git requires more time to master, but also provides a lot of built-in functions that facilitates the realization of plans. Since SVN is centralized, its users are also heavily "master host" dependent. Git as a distributed VCS does not face such issues. It is also faster and its merge implementation leaves little room for conflicts. Picking one VCS over another is only dependent on what advancement level is required for the project where it will be used.

Figure 7 Visualization of rebase operation. Snapshots changes are applied step by step to a new commit.



CI tip: A rebase operation is extremely useful for small "buffer" branches that always have to be rebased against their parent to keep pre-integrated changes and guard against conflicts met during their application. Rebase operations also help to keep the history clean.

References

[1] http://svnbook.red-bean.com/en/1.7/svn.basic.version-control-basics.html#svn.basic.repository
[2] http://svnbook.red-bean.com/en/1.7/svn.basic.in-action.html
[3] http://svnbook.red-bean.com/en/1.7/svn.ref.properties.html
[4] http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.diff.html
[5] https://git-scm.com/book/en/v2/Getting-Started-Git-Basics
[6] http://ericsink.com/vcbe/html/directed_acyclic_graphs.html
[7] http://svnbook.red-bean.com/en/1.7/svn.advanced.pegrevs.html
[8] http://svnbook.red-bean.com/en/1.7/svn.branchmerge.basicmerging.html
[9] http://svnbook.red-bean.com/en/1.7/svn.tour.cycle.html#svn.tour.cycle.resolve
[10] http://svnbook.red-bean.com/en/1.7/svn.branchmerge.advanced.html#svn.branchmerge.cherrypicking
[11] http://svnbook.red-bean.com/en/1.7/svn.branchmerge.advanced.html#svn.branchmerge.nomergedata
[12] http://svnbook.red-bean.com/en/1.7/svn.branchmerge.basicmerging.html#svn.branchemerge.basicmerging.reintegrate
[13] https://git.wiki.kernel.org/index.php/GitSvnCrashCourse
[14] https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
[15] https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes
[16] http://gitready.com/beginner/2009/01/21/pushing-and-pulling.html
[17] https://git-scm.com/book/en/v2/Git-Internals-Git-Objects
[18] https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging
[19] http://ariya.ofilabs.com/2013/09/fast-forward-git-merge.html
[20] https://sandofsky.com/images/fast_forward.pdf
[21] https://git-scm.com/book/en/v2/Git-Branching-Rebasing
[22] http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.log.html
[23] http://git-scm.com/docs/git-log

Other references worth to mention

[24] http://nvie.com/posts/a-successful-git-branching-model/
[25] http://rogerdudler.github.io/git-guide/
[26] http://think-like-a-git.net/

About the author

I work in MBB System Module (Software Configuration Management) Department. Every day I maintain and develop Linux based Continuous Integration environments which are responsible for building and testing LTE (Advanced) and WCDMA products. My position's R&D aspect allows me to constantly develop my competencies related to fully automated Continuous Delivery systems.

Marcin Gudajczyk
Senior Engineer, Software Configuration
MBB System Module

From Customer Documentation to User Experience

Tomasz Prus
Manager, Customer Documentation
MBB R&D Mgmt and Automation

NOKIA

1. Introduction

We live in a world of information. Every day we are bombed with tons of data coming from everywhere. Sounds like a cliché? Yes, it does, but that is the fact. In an era of telecommunication and mobile technologies, we see an exponential growth in data traffic. Nokia is a significant part of this global revolution. By delivering cutting edge mobile infrastructure technology, we give shape to people's daily life. Thinking of Nokia products I mean hardware, software, and customer documentation. This article is about good technical documentation, which plays a critical role in helping users to understand our products or solutions. Although the belief is that 'nobody reads documents' still exists, research studies suggest that users, regardless of their age or gender do read documentation to understand products and technology [1]. Documents are very often the only way to translate highly advanced knowledge into an understandable format – operating documentation. This is what the Nokia Customer Documentation department does. We are the bridge between our customers and Research and Development (R&D). We build this bridge together with over 300 technical writers, proofreaders, project managers, and graphics designers located all around the world. Every year we publish hundreds of operating documentation sets covering Nokia's portfolio.

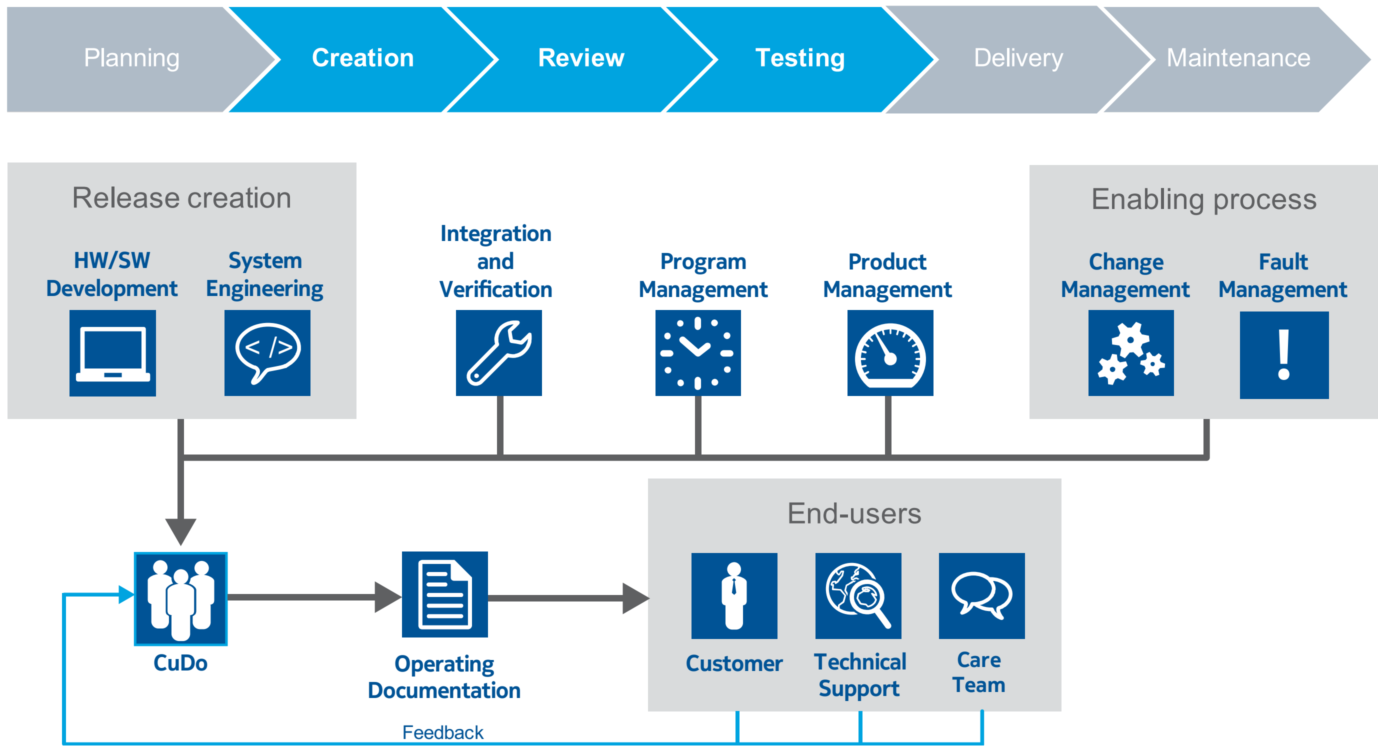
2. The way of working

The functional goal of technical content is to help people use a product successfully. Technical content may have persuasive objectives; for example, describing the additional features available in a more expensive product edition may encourage people to upgrade to it. But the informational angle is more important than the persuasive function. When managed properly, technical information can help you to:

- Meet regulatory requirements with minimum cost
- Extend your global reach by delivering content optimized for each market
- Reduce technical support costs
- Reduce product returns
- Improve customer satisfaction
- Lower the overall cost of information development [2]

Nokia Customer Documentation Department creates operating documentation for almost all Nokia products and solutions. We cooperate with different Business Units to fulfill their requirements and satisfy our customer needs. Our content creation process consists of four main steps: planning, creation, review, and testing, see **Figure 1** Customer Documentation process and main stakeholders.

Figure 1 Customer Documentation process and main stakeholders.



At each stage we work in close cooperation with various stakeholders in order to produce several types of information:

- Descriptions (e.g. feature or product descriptions)
- Procedures (e.g. installation manuals, configuration guides, commissioning or troubleshooting instructions)
- Reference (e.g. parameter, counter, and alarm descriptions)

Figure 1 presents a simplified content creation process and visualizes its main stakeholders. In the **planning** phase we project and organize our delivery by analyzing customer and R&D requirements. At this stage we make sure that all technical aspects will be covered in the content. We establish communication channels with Subject Matter Experts (SMEs) who will act as key content reviewers and input providers.

The **creation** stage is the main stage of a technical writer's work. We use different source materials such as specifications, technical analysis, marketing materials, technical reports, and reference information taken from various databases. Depending on the information type we develop pieces of information that are consulted and review by SMEs. This phase is crucial as we work in a global company where employees are spread all around the world. Gathering input for the documentation requires excellent communication and presentation skills, whereas content creation needs strong product understanding and technical skills.

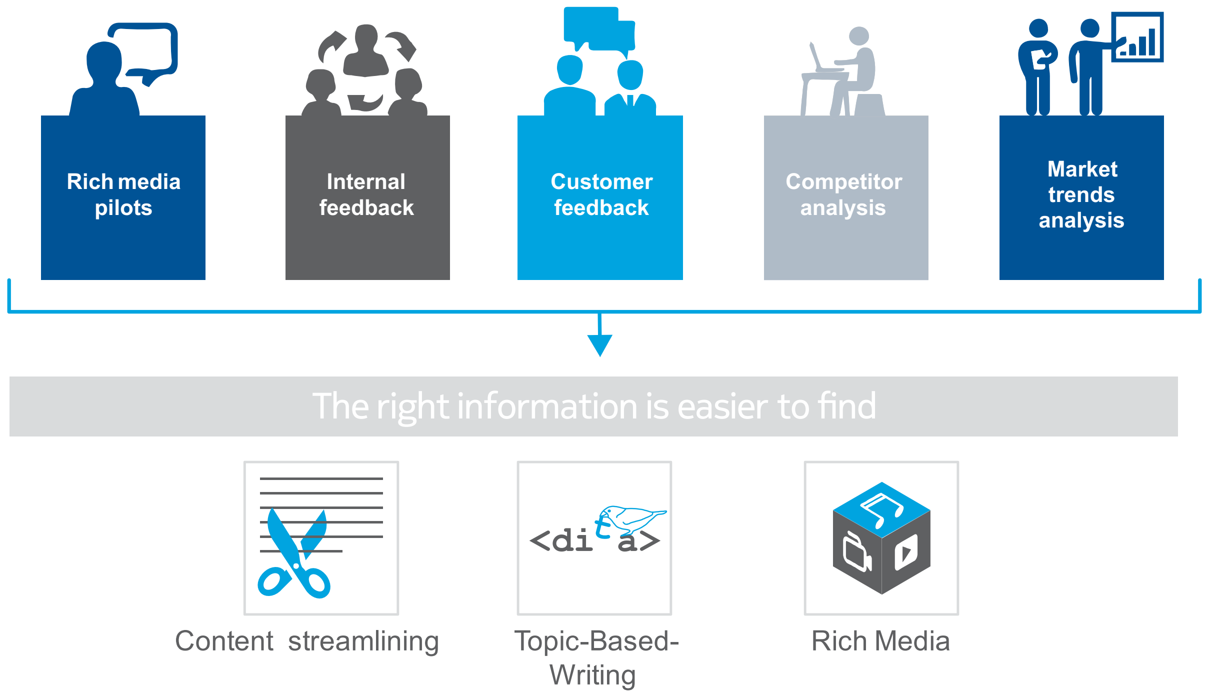
The **review** step is conducted to assure high quality standards. This stage requires cooperation between reviewers and technical writers. Technical reviews can be organized in different ways:

- Face to face meeting
- A virtual meeting (using teleconference and desktop sharing features)
- Through collaboration portals such as JIRA, SharePoint, Confluence
- Through shared review tools, where reviewers can see and comment each other's remarks
- By email

After an agreed period of time, comments and suggestions are approved or rejected and the document is updated with new content. It may happen that the review round occurs more than once. When all stakeholders have completed their reviews and all updates are made, the technical writer sends the document for testing.

Operating documentation procedures are **tested** by Integration and Verification (I&V), System Verification (SyVe), or Technical Support teams. Testers check and verify the documented procedures in a laboratory environment using relevant hardware and software configurations. When an issue is found in the documentation, testers report them in the relevant place (e.g. using dedicated tool such as Quality Center) and technical writers need to update documentation accordingly. After a final approval the document is published as part of a specific product or system documentation library to Nokia Online Services (NOLS) portal, available to our customers.

Figure 2 Nokia Customer Documentation transformation strategy.



Customer Documentation follows R&D product lifecycle milestones and contributes to almost all software and hardware deliveries. We actively participate in feature development and testing. Currently most of Nokia products are developed in Agile mode, and so is the documentation. Close cooperation with technology experts makes us more efficient. Being flexible and ready to adapt is our asset for increasing value.

As Customer Documentation we went through a very long journey. Now we are in the middle of a change of mindset. More information and more types of information are available online. Users become increasingly aware that they need more efficient ways to find the information they need. Interface and information designers, developers and technical communicators must rise to the challenge. To do this, we must employ the techniques of user-centered design to understand user's tasks and goals. We must find ways to design the user interaction in a way which matches the user's needs, which is easily learned, and which is efficient, effective and engaging to use – in other words, we must design for usability [3]. To satisfy increasing customers' demands we developed our new path and started the Customer Documentation transformation project.

3. Transformation

At the heart of every great idea is a desire to evolve. Nokia Customer Documentation strives to be on the cutting edge. We have built our transformation strategy based on market trends, competitor analysis, customer and internal feedback. We do not want to follow. We desire to create new trends in technical communication market. We decided to build a new documentation strategy on three main pillars:

- Streamlined content
- Topic-based writing
- Rich media

4. Darwin Information Typing Architecture (DITA)

Creating documentation using the traditional static file approach is very inefficient. Writers very often needed to struggle with lack of synchronization between different renditions e.g. *.doc and *.chm. DITA solves this problem.

As the first step of our transformation, we decided to migrate our content to DITA. DITA is an Extensible Markup Language (XML), open standard for structuring, developing, maintaining, and publishing the content. This is the universal format for structured documents and data on the Web [4]. XML gives us also other benefits, such as:

- Openness (XML is an open W3C standard that evolves and is continuously under development)
- Separation of form from the content (XML stores meaning, not presentation)
- Tagging system (In XML you can create your own sets of tags)

DITA introduces a complete content model, a framework that represents the structure of the information to be stored. In DITA, a content model is implemented through information types, or topic types. DITA's base content model (that is, the standard, default DITA model) defines three information kinds: concept, task, and reference. Concept, task and reference topics contain different types of content. Concept topics contain explanations, task topics contain procedural steps, and reference topics contain tabular look-up information [5].

Content is structured and stored in a neutral XML format, which can be used to directly produce a full range of outputs such as HTML, PDF, or Java Help from a Content Management System.

5. Content Management System (CMS)

Everyone is familiar with basic reuse—copying and pasting from one document to another. But copying and pasting creates disconnected copies, which then must be updated separately [2].

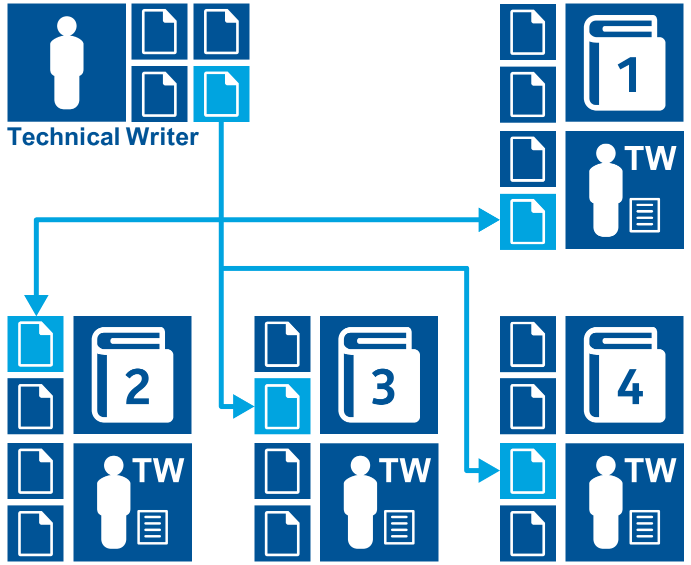
In order to efficiently use all DITA features, technical writers need to use a powerful CMS. A CMS is a platform that allows large groups to cooperate in the content creation and review phases. At first glance it is a storage tool for DITA XML files, as well as graphics and other objects that can be used to create different types of documents. Crucial feature is the integration with an XML editor like Adobe FrameMaker or oXygen. A CMS also provides versioning and branching mechanism to manage the relationships between thousands or even millions of components.

One of the main advantages of a CMS is the ability to handle content reuse. Reusing content lets you reduce content development costs while simultaneously improving the quality of the information. The technical writers' goal is to create complete topics that can be reused as such in different documents. This process is called single sourcing.

6. Single Sourcing

These days business efficiency is a key asset. In Customer Documentation we constantly work on increasing single-source authoring, which is a writing technique that allows content to be used multiple times in many places. Single sourcing can be compared to recycling, as many writers can access and use the same particular chunk of information. Single source publishing is mostly initiated by the wish to supply information from one source for a range of media in order to keep contents consistent and minimize the creation effort.

Figure 3 Example of single sourcing concept.



The advantage of this approach is that when you have to update your information, you can make a single change in a module that can be reflected in many documents at once through automatic republishing. This is much more effective and efficient than finding each document that must be updated, finding the specific information that must be changed, and making the same change over and over again, manually fixing up the formatting for each document if the change affects page breaks. The disadvantage of this approach is that you must define the rules for making changes, and that means that you must first declare ownership [6].

Single sourcing is not easy and sometimes causes communication problems, especially in big virtual teams, but the idea is always worth to consider. An important question that may be raised is 'How can I change the content strategy to gain advantages of single sourcing?'

7. Topic Based Writing and minimalism

Topic-based writing is an approach to structuring information. The basic assumption is that writing refers to the content that is chunked into short topics instead of longer pieces of information. According to its main principles, content needs to be:

- Correct. You cannot save technically incorrect information with snazzy formatting
- Relevant. Accurate information is not enough—readers want information that addresses their specific issue
- Concise. Most readers do not want to wade through huge volumes of information to find what they need
- Accessible. Readers must be able to get at the information; that means, for example, providing graphics that do not use tiny type and addressing the needs of readers with varying degrees of literacy, visual acuity, computer skills, and so on
- Usable. Readers must be able to find the information they want and understand it [2].

In practice it means that technical writers create content that user can read and understand, even when taken out from the context of the document. Topics are created as separate instances and chained by references and links. Minimalism concerns editing down content until you provide exactly the right information (and no more) for the user to perform the task or understand the concept basically, to get the information they need at the point they need it. To apply minimalism successfully, you need to understand the product very well; frequently get feedback from users; and be willing to cut out all non-essential content so that users are left with streamlined, usable content. Minimalism also means that you document the *best way* to do something not *all* the ways to do something. By removing all the extra information from the content, you are left with content where each word has value. Users have to wade through fewer topics of higher quality content. When every topic is well written, the user can find information quickly and the information they find is exactly the type of information they need. They can quickly read or even scan the topic, get their answer and be done. The result: Maximum user satisfaction [7].

8. Beyond the documentation – User Experience (UX)

When words are not enough, what then? UX comes into play. UX includes the practical, experiential, affective, meaningful and valuable aspects of human–computer interaction [8]. It is an umbrella term describing all the factors that contribute to user's overall perception of a product.

So, why is this so important for Customer Documentation? The world of information is changing. Users require more attractive, accurate, and visual ways of providing information. Documentation becomes more engaging, interactive, and easy to use. Technical writers need to consider usability, usefulness, and the aesthetic appeal of the content. The main goal of the documentation is now usability, which means that the people who use the product can do so quickly to easily accomplish their own tasks.

What makes great UX? Great UX documentation is not just about good looks. According to *uxforthemasses.com*, UX documentation needs to be:

1. Useful – First and foremost a great UX document is useful to its audience. It clearly contributes to the overall goal of the work being undertaken– whether that has to create a design, define a strategy or carry out some research
2. Appropriate – A great UX document is appropriate to its purpose, to the situation and to its audience.
3. Usable – Just like a usable interface, a great UX document is easy to understand and use. Ideally it should be as self-explanatory as possible
4. Presentable – A great UX document does not need to be a work of art, but it should be presentable
5. Accessible – A great UX document should be easily accessible. It should be easy to find, easy to access and in a suitable file format [9].

Nokia Customer Documentation found its own way to increase the User Experience. Making documentation easy to find, more engaging, and attractive is now one of our main goals.

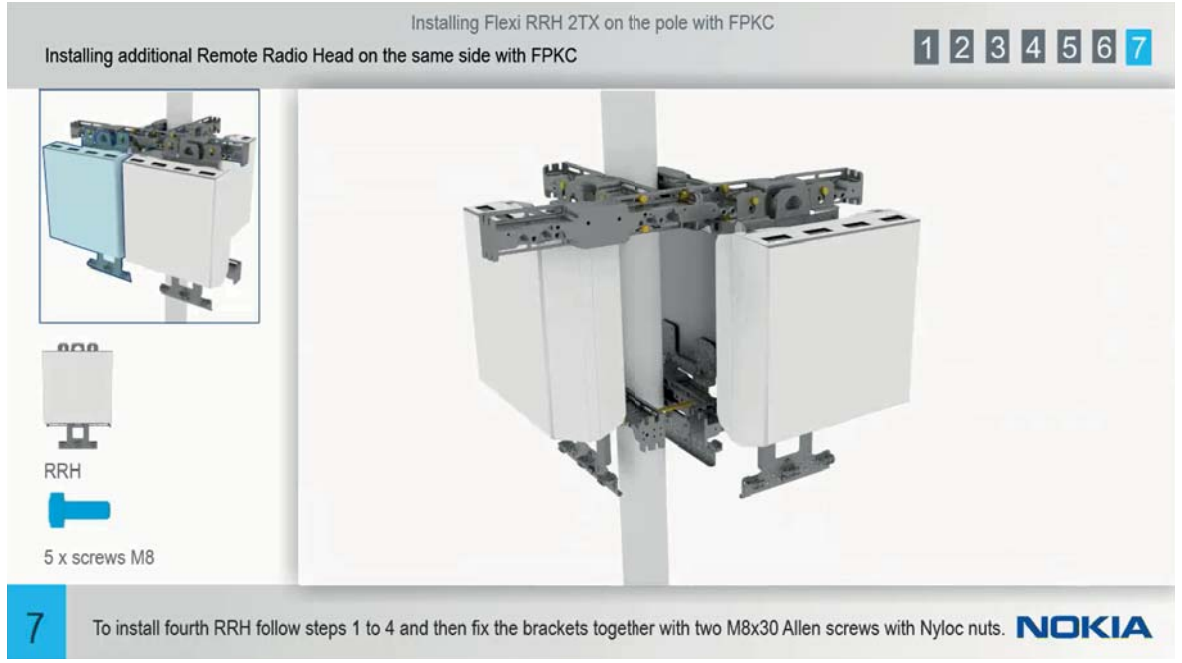
9. Rich media

Now, when more and more content is being delivered online, users can go beyond static content by consulting videos and animations. This multimedia information is often referred to as rich media. In Customer Documentation we have defined, piloted, and rolled out several types of rich media:

- Interactive graphics, showing information flows
- Videos, showing detailed (mainly hardware) procedures
- Animations, presenting complex features in a more comprehensive way
- E-learning-like solutions, giving more information about the functionality or product

Main goal of rich media is to be effective and straight to the point while presenting the information. Rich media is not only for marketing purposes anymore. There is a high demand for multimedia content in technical documentation. Using rich media to deliver information can improve customer engagement and satisfaction. This is now. How about the future? It is right behind the corner and it is called augmented reality. Customer Documentation's desire is to create a solution that allows less experienced engineers to perform e.g. troubleshooting procedure on a real life environment. However, the key driver for successful troubleshooting is the remote support like voice/video call between the field engineer and remote support expert. The key concept is to give real time instructions using e.g. Google Glasses or virtual reality.

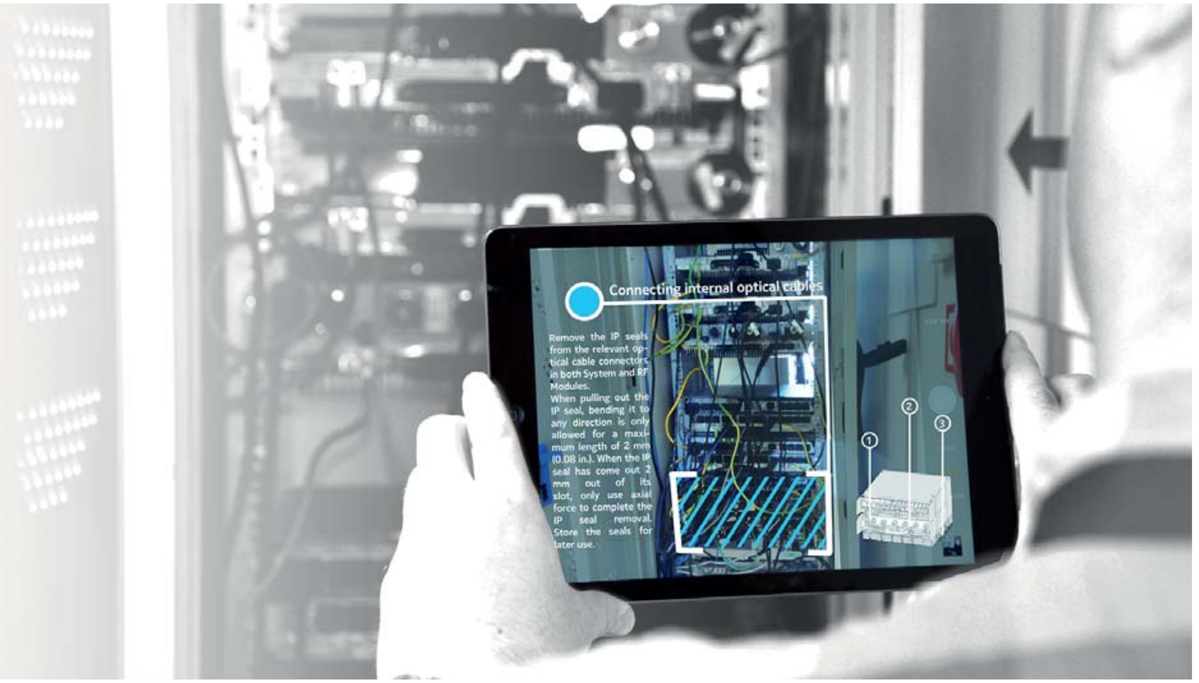
Figure 3 Example of rich media video.


Installing Flexi RRH 2TX on the pole with FPKC

Installing additional Remote Radio Head on the same side with FPKC

To install fourth RRH follow steps 1 to 4 and then fix the brackets together with two M8x30 Allen screws with Nyloc nuts.

Figure 5 Example of using augmented reality in Customer Documentation.



Connecting internal optical cables

Remove the IP seals from the relevant optical cable connectors in both System and RF Modules. When pulling out the IP seal, bending it to any direction is only allowed for a maximum length of 2 mm (0.08 in.). When the IP seal has come out 2 mm out of its slot, only use axial force to complete the IP seal removal. Store the seals for later use.

Evolution is braving new territory. By embracing new simulation technologies we can improve engineers' skills (in a similar way as aircraft pilots) before they go into the field and reduce troubleshooting costs significantly.

10. Conclusion

Historically, technical communication has been a cottage industry where each technical writer was responsible for a specific content area. Today, technical communication is moving into a manufacturing model. This approach requires a huge shift in mindset for experienced writers. Instead of owning all aspects of a book or help system, writers become content contributors who collaborate to produce a final product [2]. Technical writers these days create intelligent content, structurally rich and semantically categorized, which can be therefore automatically discoverable, reusable, reconfigurable, and adaptable [5].

Market trends and research analysis show that transforming from static documentation to streamlined, single sourcing content is the way forward. Achieving this and adding rich media to Nokia's Customer Documentation portfolio will help us to be one big step ahead.

Resources

[1] G. Hiradas, "Information desing for documentation," *Communicator,* 2014.
[2] S. S. O'Keefe and A. S. Pringle, Content Strategy 101, Transform Technical Content into a Business Asset, Scriptorium, 2012.
[3] W. Quesenbery, "On Beyond Help: User assistance and the user interface," [Online]. Available: http://www.wqusability.com/articles/on-beyond-help.html.
[4] "W3C information and Knowledge Domain," [Online]. Available: http://www.w3.org/XML/.
[5] A. S. Pringle and S. S. O'Keefe, Technical Writing 101: A Real-World Guide to Planning and Writing Technical Content, Scriptorium Publishing Services, Inc., 2009.
[6] PTC.com, "10 Secrets to a Successful DITA Implementation," PTC.com, 2007.
[7] "techwirl.com," [Online]. Available: http://techwhirl.com/getting-started-with-topic-based-writing/. [Accessed 16 July 2015].
[8] "wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/User_experience. [Accessed 16 July 2015].
[9] "UX for the masses," [Online]. Available: http://www.uxforthemasses.com/create-great-ux-documents/. [Accessed 16 July 2015].
[10] J. T. Hackos, Information Development, Indianapolis: Wiley Publishing, Inc., 2007.
[11] M. T.-S. Jörg Hennig, Technical Communication – international, Today and in the Future, Lübeck: Schmidt-Römhild, 2005.
[12] J. Hackos, Managing Your Documentation Projects, Canada: Wiley, Inc., 1994.
[13] J. K. Christopher Turk, Effective Writing, Improving scientific, technical and business communication, New York: Taylor & Francis e-Library, 1989.

About the author

I studied Electronics and Telecommunications at the Wrocław University of Technology. With six years of experience gained as a technical writer, publishing specialist, and Customer Documentation manager I know how important technical content is for overall product perception. I am fascinated by new technologies and information design. Privately, I am addicted to sport, mostly football.

Tomasz Prus
Manager, Customer Documentation
MBB R&D Mgmt and Automation

Design for Security

Maciej Kohut
Security Solution Engineer
MBB Security

This brief article provides a high-level overview of the security of the product/program in the process of its creation. Security is a constant process, which is also going through the whole process of creation of the product, like software in this case. The article may be seen as an incentive to search for deeper knowledge of security issues.

1. Main security principles

There are three main security principles of protecting a system. This is the so-called CIA Triad, which include:

- Confidentiality – guarantee that unauthorized disclosure of information is prevented
- Integrity – correctness and consistency of data stored and handled by the reliable system
- Availability – guaranteed, reliable access to the services or system for authorized users

All of those three principles must be respected during every phase of the new system/application creation process. Only with this condition fulfilled, we can try to make secure software. So, first we should have a clear understanding of what the implementing of security in products means. This is often confused, especially when security of product is perceived as a mere implementing of security features.

Let us consider an LTE example, where the communication channels are secured by means of encrypted tunnels. Such encrypted tunnels are certain security features, but deploying them does not necessarily make a solution secure due to the fact that these tunnels are terminated somewhere, for example on a BTS like in this case. So, if the BTS itself is not secured, an attacker can potentially hack the BTS and see the communication between users or terminals even if this communication is encrypted between the terminal and the Base Station. This is a case when the (unsecured) BTS confidentiality is not provided, although the encryption is applied.

A good example of threats against any IT system is briefly described by the STRIDE [11] approach. This acronym stands for:

- Spoofing identity
- Tampering with data
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privileges

Last but not least, apart from the security principles defined by the CIA Triad, there are some more that should be considered: authenticity, trustworthiness, privacy, accountability, auditability and non-repudiation.

2. Design for security as a process

Design for security is a proactive process that embraces security issues not covered by other development processes. It could be said that it integrates security into the development lifecycle, thus raising the entire security level.

In the product creation process, the goal is to prevent all the security principles from being violated, no matter if intentionally or accidentally. To help achieve this goal, there are several actions to be taken, as briefly discussed in further paragraphs.

Software developers should keep focus on security at every stage of creating an application. Security is a process just like creating/developing of the product is. Both should be running in parallel as illustrated in **Figure 1**. Early phases of the creation process, when the feature screening occurs, should also involve identification, analysis and classification of threats and risks. Simultaneously, end-user data should be identified and classified in order to make possible privacy risk assessment. When all or most features of the product are specified, and the system design phase takes place, security requirements of the product should be specified. Results of threat and risk analysis should serve as a basis for this process. Every feature has to be assessed and risks have to be prioritized. Probability and criticality of threats and risks, as well as security measures to minimize the risks have to be defined.

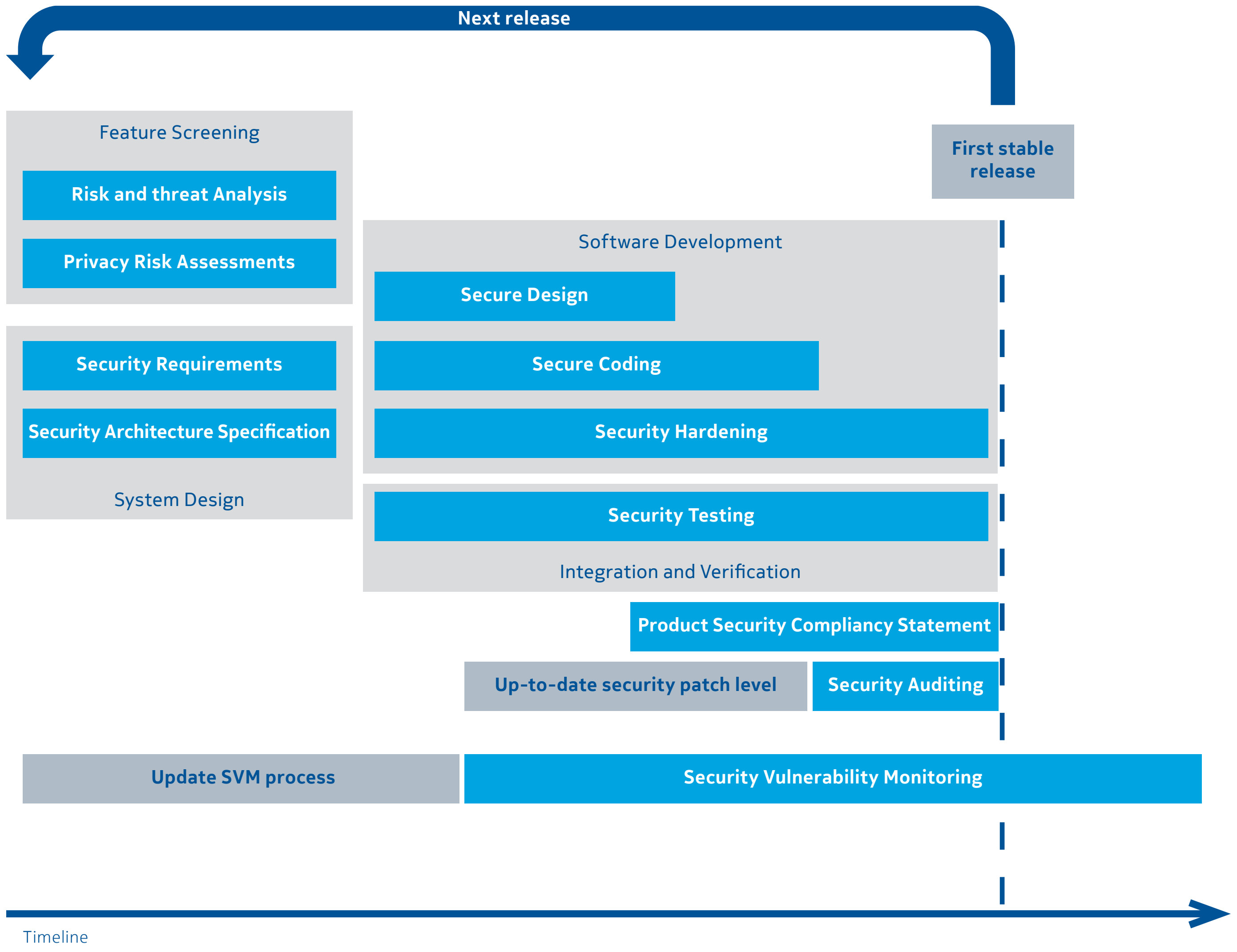
A product should be described on a high-level Security Architecture Specification to define how the security requirements are to be met. This kind of document should include a description of how all the security and privacy related features of applications, platforms, and network elements are handled. Some of those features concern data and software security, protocols, and traffic separation, as well as communication matrix.

A newly created product frequently uses third party software, which is often an open source, (for example operating systems like Linux, open databases and so on). Any security gaps in this kind of additional component are unacceptable, therefore it is crucial to have knowledge of any security vulnerabilities they might have. That is why a Security Vulnerability Monitoring (SVM) process should be triggered for third party components as early as possible, and continued throughout the whole process of creating the product. Furthermore, the process should be kept in use even after the first stable release is published.

3. Secure coding

Let us now discuss in a little bit more details the question of the secure coding phase [11]. Avoiding mistakes that could cause an exploitable vulnerability seems an obvious goal. Security hardening is one of the integral elements of secure coding. It is about fixing security holes in the code itself. There are several techniques of doing that, for example adding code that validates inputs or replacing any

Figure 1 Design for Security in the timeline.



unsafe string function calls that are buffer-size-aware. On the Internet there are many sources of information about principles of code hardening for many programming languages. The program should be robust which means that even if there are any unexpected events (for example, abnormal data input), the program will not crash itself or, what is even worse, the whole system. One of the best known examples of such an attack is "ping of death" [21]. Situations like that should be avoided at any cost. Every developer should have a sort of paranoid approach. To respect Murphy's law would not be a bad idea here. Security by obscurity should be prohibited. Treating the source code as widely open to the public is a good approach. The code should be written in a homogenous style throughout the whole project so that it is easy to read, well documented and unambiguous. Maintaining of a code written this way is more efficient and secure. The "principle of least privilege" is very important too. Every process runs with some privileges of the user or process that has triggered it. If this user or parent process runs with restricted privileges, the amount of potential damage an attacker can do is limited even if the attacker successfully hijacks the program into running malicious code.

It is very important for every developer to know the secure coding guidelines. Samples of guidelines for many programming languages can be easily found on the Internet. The Top 25 Most Dangerous Software Errors [2] can be treated as a base for research on how to make a secure code. What is no less significant is to use a SCA (Static Code Analysis) tools, which allow to find out not only typing errors, but real vulnerabilities like buffer overflows, memory leaks, uninitialized data (which could make injection of some unexpected data possible), and so on. It is possible to simulate runtime behavior to look into any possible execution path. Although using of SCA tools could be very helpful, it cannot replace human review entirely. Keep in mind that such tools are not smart enough, and can provide false positives or negatives. An example of an SCA tool for the Java and C/C++/C# is Klocwork [3].

4. Security hardening

Next phase of software development is security hardening [13]. Usually, this process relies on reducing the surface of the system vulnerability. How is that achieved? The first rule is: multipurpose system is less secure than a single-function one. Therefore, by removing unnecessary software, services, usernames, or logins, we reduce available vectors of attack. Hardening will touch all system components: operating system, services, databases, and so on. Network access should be limited to only necessary services – for example by using firewall features, access control list and similar. If possible, banners, messages of the day and other information presented by network services should be modified to make it harder to find out what version of software is running on the system. Sometimes this is not possible, especially when the service has to be compliant with specified standards or different clients. For example, in order to change open SSH server to hide information about its version, which is displayed in every connection by default, you need to modify the source of this service and recompile it. Cryptography should be used whenever possible, plain text information should not be sent throughout the network. Passwords should not be stored without strong cryptography, and should be checked whether they match the strong password policy defined. System rights should be defined and enforced by access control list or a similar mechanism. There are many guidelines on the Internet dedicated to how to harden particular systems and services. One can also use automatic tools like bastille-linux [4].

5. Security testing

Security testing [12] is part of the integration and verification phase within the new product creation process. It is a crucial part as it allows ensuring that nothing has been missed out in the previous phases. All the protection and hardening functions should be implemented by this time, and secure testing checks if this implementation is correct. It must be assured that data can be accessed only via the strictly defined interfaces and methods, not any other way. In addition, the system should not be vulnerable to attack in the production environment. It needs to be robust against failing to function correctly because of other elements which this product will be integrated with. Specified tests against security requirements and known or unknown security vulnerabilities must be performed. There are a lot of tools available that can help with automation of these tests in many areas. For example, nmap [6], NESSUS [7], OpenVAS [8], QualysGuard [14] and other security vulnerability scanners can help to automate checks if security vulnerabilities still exist. Communication matrix and firewalls or access control list could be checked with for example nmap, hping3 [9], and many more.

Another area of security testing is web application vulnerability scanning, which can be performed by means of, for example, Accunetix [10], Burp [16], or Nikto [17].

Last but not least, there are robustness testing and (distributed) denial-of-service testing. For those you can use, for example, Codenomicon [15], hping3.

After the security testing is done, there should be generated a report listing all the activities and results. This kind of document can be used in future releases as a reference.

Every product creation process should have a person appointed the product security manager. This person will be responsible for filling in a Security Statement of Compliance, which is a method of tracking implementation of requirements defined by earlier phases of Design for Security process. It is required to have a complete view of the level of security. Moreover, customer documentation should be created with respect of privacy principles.

After the product is stable and ready to be released, a security audit should be performed. Optimally, an independent third party auditor should be engaged to inspect the product and ensure compliance with Security Policy. It is recommended that this kind of security audit be performed at least once in the product lifecycle.

6. Summary

This article is only a brief overview of what the process of creating a secure application/system, or a secure product in general, looks like. It is advised that the reader keep in mind the importance of the Design for Security process. Everyone can explore the topic in a range that is closest to his or her work on the product. Be it an architect, designer, developer, tester, engineer – in the Design for Security process there is room for contribution from everybody. Nokia offers its employees the "Product Security Education Journey – Orange belt" [20] course, which deals with the topics in more detail.

References

[1] https://www.securecoding.cert.org/confluence/display/seccode/CERT+Coding+Standards

[2] http://cwe.mitre.org/top25/

[3] http://www.klocwork.com/

[4] http://bastille-linux.sourceforge.net

[5] http://www.oracle.com/technetwork/java/seccode-guide-139067.html

[6] http://nmap.org

[7] http://www.tenable.com/products/nessus-vulnerability-scanner

[8] http://www.openvas.org

[9] http://www.hping.org/hping3.html

[10] https://www.acunetix.com

[11] https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx

[10] https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Introduction.html#//apple_ref/doc/uid/TP40002415

[11] https://collaboration.int.nokia.com/sites/SecurityTechnologies/SitePages/Secure%20Coding.aspx

[12] https://collaboration.int.nokia.com/sites/ProductSecurity/SitePages/Security%20Testing.aspx

[13] https://collaboration.int.nokia.com/sites/ProductSecurity/SitePages/Security%20Hardening.aspx

[14] https://www.qualys.com/

[15] http://www.codenomicon.com/

[16] http://portswigger.net/burp/

[17] https://cirt.net/Nikto2

[18] https://collaboration.int.nokia.com/sites/ProductSecurity/SitePages/Security%20Processes.aspx

[19] https://twiki.inside.nsn.com/bin/view/SecurityWiki/DfsecFaq

[20] Product Security Education Journey – Orange belt (Courses: PSJOGS-01A, PSJOSC-01A, PSJOPO-01A, PSJOST-01A, PSJOVM-01A)

[21] https://en.wikipedia.org/wiki/Ping_of_death

About the author

I work as a Security Solution Engineer in MBB Security Department. Our R&D department creates security orchestration software for Telco Cloud. As security solution engineers we are responsible for building, maintaining, and supporting lab environment, doing research and testing security products like firewalls, IPS and so on, as well as integrating those products into the Cloud environment based on OpenStack and VMware.

Maciej Kohut
Security Solution Engineer
MBB Security

We deliver telecommunication solutions and technologies to one quarter of the world's population. We provide the world's most efficient mobile networks, the intelligence that maximizes the value of those networks, and the services that make it all work seamlessly.

Nokia has been established in Wrocław since 2000 with The European Software and Engineering Center in Wrocław. We host a wide range of projects such as Customer Experience Management & Operations Support Systems, Radio Frequency Software, System Module, Single Radio Access Network, Long Term Evolution, R&D Management and Automation, Customer Support, Liquid Core, Technology & Innovation, Value Creation Management and Security.

Over 1000
kilometers of cables
able to cover the
distance between
Wrocław
and Brussels

Laboratory of the Nokia in Wrocław in numbers – cables

R&D in Wrocław has an impact on every development stage of our wireless access technologies. Our employees contribute to each step of the processes that give shape to our telecommunication solutions: from architecture modeling, requirement building and implementation, down to system verification and component testing in our laboratory environment.

Their sophisticated equipment gives great research potential to our labs. They are optimized for testing our software against a myriad of hardware configurations, starting from applications designed for individual clients, right down to the most technologically advanced telecommunications solutions not yet available on the consumer market.

4300
sq m of space
equal to ten
basketball
courts

Laboratory of the Nokia in Wrocław in numbers – space

The foundation of Nokia's culture lies in our four clear and timeless values – Respect, Achievement, Renewal and Challenge.

They lead and help us in making the right decisions, as well as in operating within the Nokia community on a daily basis.

Respect

We treat each other with respect and we work hard to earn it from others.

Renewal

We invest to develop our skills and grow our business.

Achievement

We work together to deliver superior results and win in the marketplace.

Challenge

We are never complacent and perpetually question the status quo.

Acknowledgements

We would like to show our appreciation and thank the entire project team for their engagement and outstanding contributions in making this unique book possible:

Tomasz Feliksik, Łukasz Grządko, Grzegorz Olender, Marcela Stopińska, and Sławomir Zborowski for all their good ideas in coordinating and supervising the work.

Thomas Athelstan-Price, Daniel Barrio Fierro, Monika Berlińska, Anna Czołpiński, Joanna Pucińska, Tomasz Prus and Daniel Rowiński for their translation, proof-reading and editorial efforts.

And finally, Martyna Kosiorek, for the idea from which everything started.

Strzegomska Street 36
53-611 Wrocław
Reception 1st Floor, Green Towers B
Opening hours 8:00 – 18:00
Phone: +48 71 777 3800
Fax: +48 71 777 30 30

Strzegomska Street 54a
53-611 Wrocław
Reception Ground Floor, Wrocław
Business Park
Opening hours 8:00 – 18:00
Phone: +48 777 38 01

Bema Street 2
50-265 Wrocław
Reception Entrance A, 4th Floor
Opening hours 8:00 – 18:00
Phone: +48 71 777 41 30
Fax: +48 71 777 41 98

Lotnicza Street 12
54-155 Wrocław
Reception Ground Floor, West Gate
Opening Hours 8:00 – 18:00
Phone: +48 71 777 4002
+48 71 777 4001

e-mail: kontakt@nokiawroclaw.pl
www.nokiawroclaw.pl