Generation of Network Service Descriptors from Network Service Requirements

Navid Nazarzadeoghaz

A Thesis

in the Department

of

Electrical & Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Electrical & Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

May 2019

© Navid Nazarzadeoghaz, 2019

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Navid Nazarzadeoghaz

Entitled: Generation of Network Service Descriptors from Network Service Requirements

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____ Chair
Dr. Dongyu Qiu

____ Examiner, External
Dr. Roch Glitho (CIISE) To the Program

____ Examiner
Dr. Dongyu Qiu

____ Supervisor
Dr. Ferhat Khendek

____ Co-Supervisor
Dr. Maria Toeroe (Ericsson)

Approved by: ____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

______________ 2019___ ________________________________
Dr. Amir Asif, Dean
Gina Cody School of Engineering and Computer Science

ABSTRACT

Generation of Network Service Descriptors from Network Service Requirements

Navid Nazarzadeoghaz, M.A.Sc.

Concordia University, 2019

Network Function Virtualization (NFV) is a new paradigm in Network Service (NS) provisioning. European Telecommunications Standards Institute (ETSI) proposed and standardized an architectural framework for NFV. By leveraging virtualization and Software-Defined Networking (SDN) technologies, NFV decouples network functionality from hardware infrastructure. This enables the automated provisioning of NSs and reduces the capital and operational costs for service operators. NFV Management and Orchestration (NFV-MANO) is a functional block in the NFV framework, and it is responsible for the deployment and life-cycle management of NSs. With NFV, the telecommunication industry is moving towards zero-touch, i.e. automation of all the processes. In order to orchestrate and manage an NS, NFV-MANO requires the NS's deployment template. This template is referred to as NS Descriptor (NSD) and contains all the details for deployment and orchestration of the NS. Designing such a descriptor requires the design of the NS, which is actually out of the NFV scope. Traditionally, service operators' experts design NSs and NSDs. However, this design activity is time-consuming and error-prone; moreover, it is not fitting the Telecom's vision of zero-touch.

In this thesis, we will propose an approach to automate the process of NS and NSD design. The approach starts from a set of requirements provided as Network Service Requirements (NSReq). The NSReq describes the required network service at a high level of abstraction and focuses on the functional, architectural, and non-functional characteristics. With the help of an

ontology representing the knowledge from Telecom standards and previous successful experiences, we decompose the NSReq. We select the set of Virtual Network Functions (VNF) from a catalog to design the NS. Considering all the levels of decomposition and the VNF's dependencies captured from the ontology, we design all the possible forwarding graphs that can form an NS. We design each forwarding graph through different steps at different abstraction levels, i.e. functional, architectural, and VNF levels. According to each forwarding graph, we design an NSD along with the traffic flows in the NS. We refine each NSD by dimensioning its VNFs using the non-functional requirements in the NSReq. Accordingly, we refine the deployment flavor of each NSD. We have developed a prototype tool as a proof of concept for our proposed approach which we will discuss later in this thesis.

Acknowledgments

I would greatly thank Dr. Ferhat Khendek for granting me the opportunity to conduct academic research under his supervision. His knowledge, patience, and belief in me were the key enablers for the accomplishment of this thesis.

I am truly grateful to Dr. Maria Toeroe (Ericsson Canada) for exposing me to her immense and valuable knowledge. Without her support, guidance, and insight it was impossible for me to be able to carry out this research and finish this thesis.

I would also thank all my colleagues at MAGIC, who were such magical friends and gave me the courage to move forward in this challenging path.

At last, I thank my family, especially my beloved parents, from the bottom of my heart for their endless love and immeasurable support in every situation and aspect.

This work has been conducted within the NSERC/Ericsson Industrial Research Chair in Model-Based Software Management, which is supported by Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson and Concordia University.

Table of Contents

List of Figures x

List of Tables xii

List of Acronyms xiii

1 Introduction 1

1.1 Network Services and Network Function Virtualization 1

1.2 Thesis Motivations 2

1.3 Thesis Contributions 3

1.4 Thesis Organization 4

2 Background and Related Work 5

2.1 Network Services 5

2.1.1 IP Multimedia Subsystem (IMS) 5

2.1.2 Voice over IP (VoIP) 8

2.1.3 Voice over LTE (VoLTE) 8

2.2 NFV Framework 9

2.2.1 Network Services in NFV: Concepts and Terminologies 9

2.2.2 NFV Architecture and Functional Blocks 10

2.2.3 NFV Information Elements 14

2.3 Model Driven Development (MDD) 15

2.3.1 Unified Modeling Language (UML) 15

2.3.2 Model Transformation 16

2.4 Related Work.....16

3 The Modeling Framework.....20

3.1 Network Service Requirements (NSReq).....20

3.1.1 Functional Requirements (FR).....21

3.1.2 Architectural Requirements (AR).....22

3.1.3 Service Access Point Requirements (SAPR).....22

3.1.4 Non-functional Requirements (NFR).....23

3.2 Network Function Ontology (NFO).....24

3.2.1 Functionalities.....25

3.2.2 Architectural Blocks (AB).....27

3.2.3 Service Access Points (SAP).....29

3.2.4 Context.....30

3.3 VNF information elements.....31

3.3.1 VNF Descriptor (VNFD).....31

3.3.2 VNF Architecture Descriptor (VNFAD).....34

3.3.3 VNF Package Info.....37

3.4 VNF Catalog.....38

3.5 Protocol Stack.....38

3.6 Solution Map (SM).....38

3.7 Network Service Descriptor (NSD).....40

3.7.1 NsVLD.....41

3.7.2 SAPD 41

3.7.3 VNFFGD 41

3.7.4 NsDf........ 42

4 Derivation of Network Service Descriptor from Functional and Architectural Requirements 44

4.1 Overall approach 44

4.2 Steps of the Approach 46

4.2.1 Step 1 – Initialization of the SM model........ 46

4.2.2 Step 2 – Decomposing the SM model 48

4.2.3 Step 3 – Selecting the VNFs........ 57

4.2.4 Step 4 – Generation of forwarding graphs........ 58

4.2.5 Step 5 – Generation of Network Service Descriptors........ 62

4.2.6 An example of NSD generation........ 73

4.2.7 Update of the Network Function Ontology 76

4.3 Limitations 81

5 Network Service Descriptor Refinement (w.r.t Non-functional Requirements) 83

5.1 Overall approach 83

5.2 Steps of the Approach 84

5.2.1 Propagating the NFRs........ 84

5.2.2 Dimensioning the VNFs 87

5.2.3 Tailoring the NS Deployment Flavor 92

5.3 Limitations 95

6 Prototype Tool 96
6.1 Prototype architecture 96
6.1.1 Transformation structure 98
6.1.2 Challenges in using the ATL language 101
6.2 Case study: VoLTE service using IMS architecture 102
6.2.1 NSReq 102
6.2.2 NF Ontology 103
6.2.3 VNF Catalog 107
6.2.4 Protocol Stack 112
6.2.5 Results of the NSD generation and refinement process 112
6.3 Discussion 126
7 Conclusion and Future Work 127
7.1 Conclusion 127
7.2 Potential Future Work 128
8 Bibliography 130

List of Figures

Figure 2.1 - IMS reference architecture (from 3GPP TS 23.228 [13])....................................6

Figure 2.2 - NFV reference architectural framework (from ETSI GS NFV 002 [1]).............11

Figure 2.3 - NSD overview (from ETSI GS NFV-IFA 014 [9]) 14

Figure 3.1 - The NSReq metamodel 21

Figure 3.2 – An example of NSReq model 24

Figure 3.3 - The NF Ontology metamodel..25

Figure 3.4 - An example of NFO model 30

Figure 3.5 - The VNFD metamodel 32

Figure 3.6 - The VNFAD metamodel 35

Figure 3.7 - The VNF Package Info metamodel..37

Figure 3.8 – The VNF Catalog metamodel..38

Figure 3.9 – The Protocol Stack metamodel..38

Figure 3.10 - The Solution Map metamodel..39

Figure 3.11 - The NSD metamodel..40

Figure 4.1 - The overall picture of the NSD generation process 45

Figure 4.2 - The steps of the NSD generation process and the information flow...................46

Figure 4.3 – Overall view of decomposing the SM model 49

Figure 4.4 – An SM model example...73

Figure 4.5 – Pre-VNFFG example with further details 74

Figure 4.6 - NF Ontology updating case 1 example 79

Figure 4.7 - NF Ontology updating case 2 example 80

Figure 6.1 - A flowchart for the transformations in the prototype tool 97

Figure 6.2 - An example of an ATL lazy rule in the prototype 99

Figure 6.3 - An example of an ATL helper in the prototype 99
Figure 6.4 - An example of a Main rule in the prototype 100
Figure 6.5 - An overview of the transformations' structure in the prototype 101
Figure 6.6 – The NSReq model for the case study 103
Figure 6.7 – The NFO model for the case study (functional portion) 104
Figure 6.8 – The NFO model in the case study (architectural portion) 106
Figure 6.9 – The VNF Catalog model in the case study...107
Figure 6.10 – The P-CSCF VNFD model for the case study..108
Figure 6.11 – The P-CSCF VNFAD model for the case study...109
Figure 6.12 – The P-CSCF VNF Package Info model in the case study 112
Figure 6.13 – The Protocol Stack model in the case study...112
Figure 6.14 – The SM1 generated from the Transformation 1 113
Figure 6.15 - A portion of the SM2 generated from the Transformation 2 114
Figure 6.16 – The FGs portion of the SM3 generated in Transformation 3 117
Figure 6.17 – The NSD models generated in Transformation 4...118
Figure 6.18 – The NSD model for the typical IMS composition form Transformation 4.....120
Figure 6.19 - The Propagation Flows in the SM4 from Transformation 4 121
Figure 6.20 – A portion of the NsDf in the refined NSD from Transformation 6................126

List of Tables

Table 4.1 - Propagation Flow design example..76

Table 5.1 - NFR propagation example...87

Table 5.2 - VNF dimensioning example...92

Table 6.1 - The details of the architectural dependencies in the NFO..................................107

Table 6.2 - The VNF Interface elements of the VNFs in the typical IMS composition........110

Table 6.3 - The Flow Transformation elements of the VNFs in the typical IMS composition 111

Table 6.4 – The AFGs generated in Transformation 3 115

Table 6.5 - Details of the Propagation Flows in the case study..122

Table 6.6 - Details of the NFR propagation in the case study 124

Table 6.7 - Details of dimensioning the VNFs in the case study..125

List of Acronyms

3G	Third Generation
3GPP	Third Generation Partnership Project
AB	Architectural Block
ADep	Architectural Dependency
AFG	Architectural Forwarding Graph
AR	Architectural Requirement
AS	Application Server
ATL	ATLAS Transformation Language
COTS	Commercial-Off-The-Shelf
CPD	Connection Point Descriptor
CSCF	Call Session Control Function
DSML	Domain-Specific Modeling Language
EM	Element Management
EPS	Evolved Packet System
ETSI	European Telecommunications Standards Institute
FFG	Functional Forwarding Graph
FR	Functional Requirement
HSS	Home Subscriber Server
I-CSCF	Interrogating CSCF
IMS	IP Multimedia Subsystem
IP	Internet Protocol
ISC	IMS Service Control
IU	Instance Utilization
LTE	Long-Term Evolution
M2M	Model-To-Model
M2T	Model-To-Text
MCS	Maximum Concurrent Sessions
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MGCF	Media Gateway Control Function
MGW	Media Gateway
MRF	Media Resource Function
NAS	Network Attached Storage
NF	Network Function
NFO	Network Function Ontology
NFP	Network Forwarding Path
NFPD	Network Forwarding Path Descriptor
NFR	Non-Functional Requirement
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NFVI-PoP	NFVI Point of Presence
NFV-MANO	NFV Management and Orchestration
NFVO	NFV Orchestrator

NS	Network Service
NSC	Network Service Chaining
NSD	Network Service Descriptor
NSReq	Network Service Requirement
NsVl	Network Service Virtual Link
NsVLD	Network Service Virtual Link Descriptor
OCL	Object Constraint Language
OMG	Object Management Group
OSS/BSS	Operations Support System/Business Support System
P-CSCF	Proxy CSCF
PNF	Physical Network Function
QoS	Quality of Service
QR	QoS Requirement
RPS	Request Per Second
RTP	Real-Time Transport Protocol
SA Forum	Service Availability Forum
SAP	Service Access Point
SAPD	Service Access Point Descriptor
SAPR	Service Access Point Requirement
S-CSCF	Serving CSCF
SIP	Session Initiation Protocol
SLF	Subscriber Location Function
SM	Solution Map
SOA	Service Oriented Architecture
UE	User Equipment
UML	Unified Modeling Language
Vdu	Virtualization Deployment Unit
VduCpd	Vdu Connection Point Descriptor
VIM	Virtualized Infrastructure Manager
VL	Virtual Link
VLD	Virtual Link Descriptor
VM	Virtual Machine
VNF	Virtual Network Function
VNFAD	VNF Architecture Descriptor
VNFC	VNF Component
VNFD	VNF Descriptor
VnfExtCp	VNF External Connection Point
VNFFG	VNF Forwarding Graph
VNFFGD	VNFFG Descriptor
VNFM	VNF Manager
VnfVl	VNF Virtual Link
VoIP	Voice Over IP
VoLTE	Voice Over LTE

Chapter 1

Introduction

In this chapter, we introduce the research domain, the motivations and the contributions of this thesis.

1.1 Network Services and Network Function Virtualization

A Network Service (NS) is an interconnection of network functions to provide a desired functionality/behavior [1, 2, 3]. Voice over IP and video telephony are examples of NSs. A network function (NF) is a functional block with well-defined functionality and external interfaces [3]. Authentication server, firewall, and router are examples of NFs. The process of NS design consists of selecting NFs and arranging them together to achieve a composite functionality/behavior. This process is complex and error-prone, as we should select the appropriate NFs that can be interconnected and arranged according to the requirements.

In Telecom, traditionally, NSs are designed and deployed using physical NFs, i.e. NFs with dedicated hardware. Nowadays, Telecom is moving towards virtualization [4], i.e. decoupling services from hardware. European Telecommunications Standards Institute (ETSI) [5] consists of many organizations. One of them is ETSI Network Function Virtualization (NFV) Industry

Specification Group (ISG) [6] which proposed and standardized a framework for the management and orchestration of NSs using virtualization technology. They refer to this framework as Network Function Virtualization [1, 7]. NFV decouples the NFs from the infrastructure by using the NFs implemented as software and deploying them on the standard virtualized infrastructure [8]. These NFs are called Virtual Network Functions (VNFs) [2]. NSs in the NFV framework are mainly composed of VNFs and other NSs [9]. They can be partially composed of physical NFs (PNFs) [3] as well [9]. NFV Management and Orchestration (NFV-MANO) [8] is a functional block in the NFV framework, and it is responsible for the deployment and life-cycle management of NSs.

1.2 Thesis Motivations

NFV-MANO requires a descriptor for every element it manages and orchestrates, including NSs [8]. A Network Service Descriptor (NSD) [8] is a template describing the characteristics of an NS with respect to virtualization. An NSD defines the elements of the NS and specifies the deployment requirements for the infrastructure [10, 9]. NFV-MANO uses the NSD of an NS for its deployment and lifecycle management [8].

Operating and Business Support Systems (OSS/BSS) [1] is the division of an NS operator that is responsible for NS and NSD design in addition to other management tasks. Usually, an expert or a group of experts is responsible for the design of NSs and NSDs according to NS requirements. This is done manually. As discussed earlier, NS design is a complex task. Designing an NSD manually is also a tedious and error-prone task since it requires the knowledge of many details and the manipulation of many attributes. Automating the process of NS and NSD design is highly desirable.

For a given network service requirement and a set of available NFs, there might be multiple NS (and NSD) solutions. Among these solutions, one or some may be more efficient than others. Exploring all solutions manually to find the most efficient one is complex. NS design automation enables this exploration and the possibility of optimization.

In this thesis, our goal is to propose a method to automatically design NSs and NSDs from available VNFs starting from network service requirements. Our work was inspired by [11], in which the authors have proposed a method for decomposing user requirements and designing applications automatically from commercial off-the-shelf components. There are discrepancies between these two works since they focus on different domains. In our work, for instance, we need to design the traffic flows inside the network services which is an important aspect, and it is not considered in [11].

1.3 Thesis Contributions

The contributions of this thesis are as follows:

1) A model-driven approach for the automated design of network services from high level network service requirements. This method is based on a network service knowledge-base, referred to as ontology. The network functions of the network service are selected from the available VNFs described in a VNF catalog – PNFs or nested network services are not supported. The method generates all the possible NSDs and conforms to the ETSI NFV standards [10, 9].
2) A model-driven method for the automated refinement of the generated NSDs according to the non-functional requirements. In this method, we dimension the VNFs of the NSDs and calculate the capacity of the links connecting them. Finally, we refine the NSDs' deployment flavors.

3) The definition of an ontology (knowledge-base) for designing network services, and a method to enrich it based on new information provided in network service requirements, and new VNFs added to the catalog.
4) A prototype tool for all the aforementioned methods. The tool is demonstrated with the VoLTE service [12] using IMS architecture [13] case study.

1.4 Thesis Organization

The thesis is composed of seven chapters. In Chapter 2, we provide the background on network services, NFV framework, model-driven engineering, and we discuss related work. In Chapter 3, we discuss the modeling framework on which our methods are based. It includes all the metamodels we propose and the metamodels developed in the ETSI NFV standards [10, 9, 14]. In Chapter 4, we discuss our approach for the design of network services and NSDs with respect to the required functionality and architectural constraints. In this chapter, we also discuss the method for enriching the ontology. In Chapter 5, we describe the method for refining the network service descriptors with respect to the non-functional requirements. In Chapter 6, we present the prototype tool we developed along with the case study used to demonstrate its usage. In Chapter 7, we conclude the thesis and discuss potential future work.

Chapter 2

Background and Related Work

In this chapter, we introduce the concept of network services and three well-known examples, IP Multimedia Subsystem (IMS) [13], Voice over IP (VoIP) [15], and Voice over LTE (VoLTE) [12]. Moreover, we discuss NFV and its architecture [1] in more depth, and we briefly introduce the Model Driven Development (MDD) paradigm [16]. Finally, we review the work related to this thesis.

2.1 Network Services

A network service is a composition of network functions with a specific arrangement to provide a composite functionality/behavior. An NS may be composed of other NSs referred to as nested NSs. Such an NS is called composite NS [3]. An NS has specified traffic flows which traverse the NFs from one endpoint to another. In other words, these traffic flows are end-to-end inside the network service [17]. The environment can access a network service from its endpoints. IMS [13], VoIP [15], and VoLTE [12] are three well-known network services.

2.1.1 IP Multimedia Subsystem (IMS)

Third Generation (3G) networks [18] goal was to merge the cellular networks with the Internet [19]. Therefore, cellular network users can access the Internet and its services such as

VoIP and conferencing. IMS is an element in the 3G architecture which provides ubiquitous cellular access to Internet services. IMS has three major contributions in 3G including guaranteed Quality of Service (QoS), flexible charging, and service integration [19].

Real-time multimedia services offered on the Internet follow the best-effort model, i.e. the packets order, bandwidth, and delays are not guaranteed [19, 20]. IMS provides a predictable user experience for such services by establishing synchronized sessions with QoS provisioning. By using IMS, operators are able to use different charging models for different services, e.g. flat-rate, time-based, and QoS-based. IMS defines standard interfaces that enable operators to integrate their services with services from third-party operators. Therefore, they can provide their users with new and multi-vendor services [19].

2.1.1.1 *IMS Architecture*

Third Generation Partnership Project (3GPP) [21] is a group that has standardized IMS. It has proposed the IMS reference architecture in [13] which is showed in Figure 2.1.

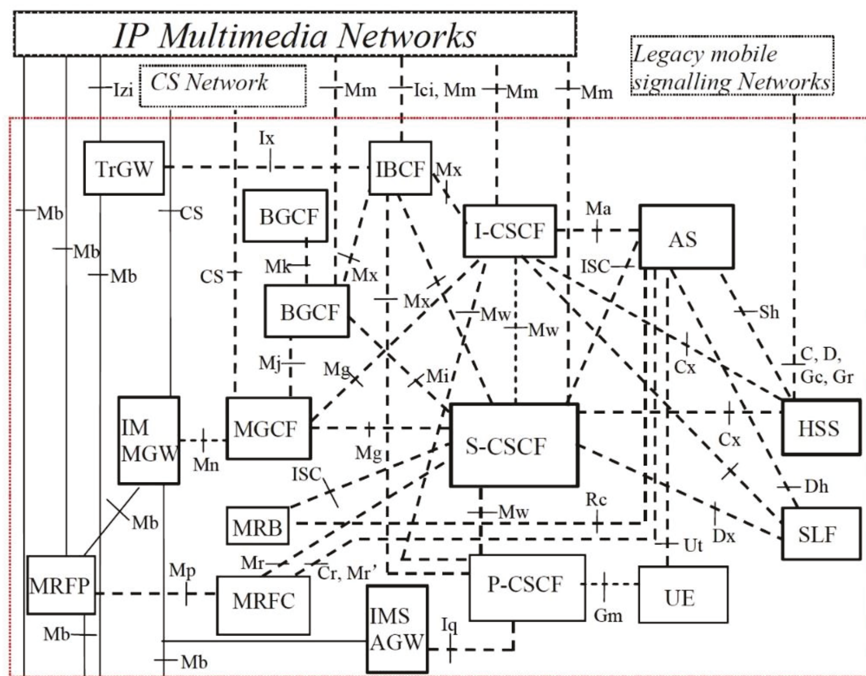

Figure 2.1 - IMS reference architecture (from 3GPP TS 23.228 [13])

The main functional blocks in this architecture are Home Subscriber Server (HSS), Call/Session Control Function (CSCF), Application Server (AS), Media Resource Function (MRF), Media Gateway (MGW), and Media Gateway Control Function (MGCF).

HSS: It stores the users' subscription information used to handle their multimedia sessions. This information includes location information, authentication and authorization information, user profile, etc. An IMS network may have more than one HSS according to the number of its subscribers. In such a case, a Subscription Locator Function (SLF) is required to find the right HSS for a user [19].

CSCF: It is the core function that processes the Session Initiation Protocol (SIP) [22] signaling in the IMS network [19]. IMS uses this protocol to establish and manage multimedia sessions over IP networks. CSCF has three different blocks including Proxy-CSCF (P-CSCF), Interrogating-CSCF (I-CSCF), and Serving-CSCF (S-CSCF) [19].

P-CSCF is the first contact point between the terminal and the IMS network. The terminal is where the User Equipment (UE) is connected to. It receives all the SIP requests from the terminal. It forwards these requests and the generated responses towards the appropriate direction, i.e. towards the terminal or IMS network [19]. **S-CSCF** is the central node for the SIP signaling. It is essentially a SIP server, meaning it responds to the SIP requests. It performs the session control function, i.e. setting up and termination of sessions. It is the SIP registrar as well, i.e. gives the users SIP address, and keeps the binding between their IP and SIP addresses [19]. **I-CSCF** assigns an appropriate S-CSCF to a user according to the information it retrieves from the HSS. It also forwards the SIP requests/responses to the assigned S-CSCF [23].

AS: It is the function that hosts and executes the SIP applications [19], e.g. Voice over IP, conferencing, etc. It receives/sends SIP requests for its service from/to S-CSCF [23].

MRF: It provides functionalities for the media resources including mixing media streams, transcode between different CODECs, obtain statistics and media analysis, etc. [19].

MGW: It is the interface of IMS to circuit-switched networks for the media stream. It sends and receives the IMS media stream over the Real-Time Transport Protocol (RTP) [24]. It also transcodes the stream when the IMS does not support the CODEC used by the circuit-switched network [19].

MGCF: It is responsible for the protocol conversion between the SIP and the call control protocols in circuit-switched networks. It also controls the resources in the MGW [19].

2.1.2 Voice over IP (VoIP)

All the voice communication services using the Internet Protocol (IP) technology instead of the circuit-switched technology are referred to as VoIP. The functionalities in a VoIP network are similar to the circuit-switched networks. They include Media Gateway and Media Gateway Controller functions [15].

Media Gateway is an interface for the voice content transportation over the IP networks. It is responsible for call originating, call detection, analog-to-digital voice conversion, and CODEC functions. Media Gateways use RTP protocol. Media Gateway Controller is mostly responsible for the call signaling coordination of the voice calls. It uses SIP or H.323 [25] protocols. A VoIP service can be realized using IMS, as IMS provides functionalities required for VoIP services [15].

2.1.3 Voice over LTE (VoLTE)

Voice over LTE is a VoIP service based on Long Term Evolution (LTE) [26] technology. LTE is based on IP networks and does not support the circuit-switching technology [12]. As discussed earlier, VoIP partly uses circuit-switched networks. Therefore, to have voice and

video communication on LTE VoLTE service was developed. To develop VoLTE, IMS has been adopted in the cellular networks to provide high-quality voice services. VoLTE voice traffic is delivered from the UE to the Internet through the gateways of Evolved Packet System (EPS) [27]. VoLTE signaling traffic is also delivered from the UE to IMS through EPS [12].

2.2 NFV Framework

2.2.1 Network Services in NFV: Concepts and Terminologies

An NS in the NFV framework is a composition of VNFs with specified or unspecified connectivity. The NSs with specified connectivity has one or multiple forwarding graphs and provide certain functionality/behavior [1, 3, 9]. The NSs can be partially composed of PNFs and have nested NSs as well. A VNF is a software implementation of an NF that can be deployed on NFV infrastructure [3]. In an NS with specified connectivity, the VNFs are interconnected with logical connections referred to as Virtual Links (VL). A VL defines the connectivity between connection points and the connectivity's performance characteristics, e.g. bandwidth, latency, etc. [3].

A VNF exposes one or multiple logical ports to communicate with other NFs or the environment. These ports are referred to as VNF External Connection Points (VnfExtCp). VLs interconnect VNFs through their VnfExtCp [2, 14]. A VNF is composed of components called VNF components (VNFC) with specific connectivity [2]. The VNFCs in a VNF are interconnected via VLs. The VLs connecting the VNFs are called Network Service Virtual links (NsVl) and the ones connecting the VNFCs are called VNF Virtual Links (VnfVl) [10]. A VNFC has one or multiple logical ports called VNFC Connection Points (VnfcCp). VnfVls connect the VNFCs through their VnfcCps. A VnfExtCp exposes one or multiple VnfcCps in the VNF to the outside [10].

VLs may have one of the three flow patterns including E-Line, E-LAN, and E-Tree. An E-Line VL has only two connections, i.e. it can connect only two connection points (VnfExtCp/VnfcCp). An E-LAN has more than two connections, and it provides a fully connected mesh between all of its connections. An E-Tree VL has multiple connections, one as the root and the rest as the leaves. The root communicates with the leaves, but the leaves cannot communicate with each other [10, 28].

A forwarding graph in an NS in the NFV framework is referred to as VNF Forwarding Graph (VNFFG). A VNFFG is a logical graph defining the connectivity between the NS network functions, i.e. VNFs/PNFs/nested NSs, and the NS connection points [3, 1]. The main purpose of a VNFFG is to define the traffic flows between the NFs and connection points [3]. An NS exposes one or multiple VnfExtCps of its VNFs to the environment through logical ports referred to as Service Access Points (SAP) [10]. The VNFFG specifies the NS traffic flows using Network Forwarding Paths (NFP). An NFP is a sequence of VnfExtCps and SAPs through which the packets traverse according to different routing policies [10, 9].

2.2.2 NFV Architecture and Functional Blocks

In [1], ETSI NFV proposed the NFV architectural framework shown in Figure 2.2. It is composed of four main modules including the NFV-MANO, VNFs and EMs, the NFV Infrastructure (NFVI), and the OSS/BSS.

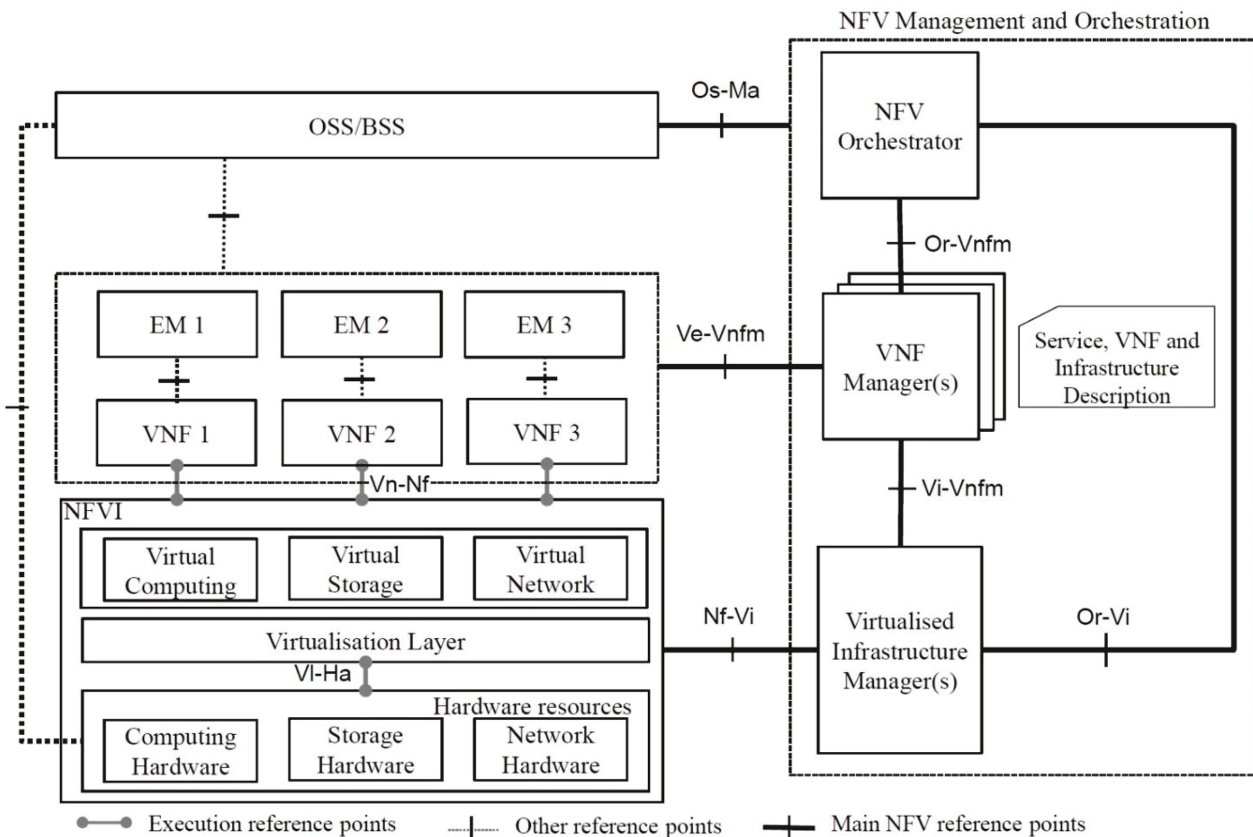

Figure 2.2 - NFV reference architectural framework (from ETSI GS NFV 002 [1])

2.2.2.1 *NFV Management and Orchestration (NFV-MANO)*

NFV-MANO module is responsible for managing the infrastructure, allocating required resources to the VNFs and NSs, and orchestrating them [8]. As it is shown on the right side of Figure 2.2, it has three main functional blocks including NFV Orchestrator (NFVO), VNF Manager (VNFM), and Virtualized Infrastructure Manager (VIM). In the following, we discuss the non-exhaustive list of responsibilities for each of these blocks.

- ***NFVO:*** According to [8], the responsibilities of NFVO go into two categories, including NS lifecycle management, and NFVI resources orchestration. In the former category, NFVO is responsible for on-boarding the NSs. This process includes verification of the integrity and authenticity of the NS deployment template and its elements. In this category, NFVO is

also responsible for the NS instantiation and lifecycle management, e.g. update, query, scaling, performance monitoring, and termination. In the latter category, NFVO is responsible for validating and authorizing the NFVI resource requests from the VNFM. It also manages the policies for resource access control and allocation, placement optimization, etc.

- ***VNFM:*** It is responsible for lifecycle management of one or multiple VNF instances. Each VNF instance should be managed by one VNFM. VNF lifecycle management includes instantiation, scaling, update and upgrade, and termination of the VNF [8].
- ***VIM:*** It is responsible for managing the NFVI resources including computing, storage, and network resources. It includes allocation, upgrade, release, and reclamation of these resources, and also managing the association between the physical and virtual resources. Its other responsibilities include virtual resources capacity management, software images management, supporting the VNFFG management, and collecting performance and fault information [8].

In addition to these functional blocks, NFV-MANO includes NS and VNF catalogs, and NFV instances and NFVI resources repositories. The first two catalogs are repositories for the deployment template of on-boarded NSs and VNFs, respectively. NFV instances repository keeps the information for all the VNF and NS instances. NFVI resources repository keeps the information for the NFVI resources and their states, i.e. available, reserved, and allocated [8]. All or some of the NFV-MANO functional blocks have access to these repositories in order to fulfill their tasks.

2.2.2.2 *Operation Support System/Business Support System (OSS/BSS)*

OSS/BSS is a proprietary part of each network service operator. It provides functions for supporting the operation and business of the operator, and therefore it is out of the NFV framework scope [8]. However, in order to manage and orchestrate network services, NFV-MANO

interacts and exchanges information with the OSS/BSS. This interaction is through the Os-Ma interface which is defined in [8].

2.2.2.3 *VNFs and Elements Managements (EM)*

As shown in the middle of Figure 2.2, VNFs are on top of the virtual resources of the NFV infrastructure. These VNFs, in addition to all other NSs' elements are deployed on these resources. Each VNF is associated with an Element Management (EM). An EM is responsible for FCAPS management functionality for the VNF. It includes fault management, configuration, accounting, collecting performance measurement results, and security management [8].

2.2.2.4 *NFV Infrastructure (NFVI)*

NFVI is the totality of the hardware and software components that create an environment enabling the deployment, management, and execution of VNFs. The hardware components can be at one point of presence (NFVI-PoP) or span across multiple NFVI-PoPs. NFVI has three components, including hardware resources, virtualized resources, and the virtualization layer [1].

Hardware resources: These are physical components that provide computing, storage, and network. In NFVI, off-the-shelf computing hardware is used as computing resources. Storage resources can be whether the storage of the servers or shared network-attached-storage (NAS). Network resources are usually routers and links providing the switching function between the computing and storage resources. The switching is whether inside an NFVI-PoP or between multiple NFVI-PoPs [1].

Virtualization layer and virtualized resources: The virtualization layer abstracts the physical resources and provides virtualized resources for the VNF deployment. Therefore, it decou-

ples the VNF software from the hardware resources and ensures a hardware independent lifecycle management for the VNF. It provides virtual computing and storage resources in the form of virtual machines (VM) [1]. Each VNFC is deployed only on one VM [2]. The virtualization layer provides the virtual network resources in the form of virtualized network paths using technologies like VLAN, VPLS, etc. The virtualized network paths provide connectivity between the VMs [1], and they realize the VLs.

2.2.3 NFV Information Elements

NFV-MANO requires different information elements for NSs and their elements in order to orchestrate and manage them. These information elements include deployment templates (descriptors) and information of instances (records). A descriptor describes the requirements and attributes for deployment and lifecycle management of an entity, e.g. a VNF or NS [8]. As discussed earlier, descriptors of on-boarded VNFs and NSs are stored in the VNF and NS catalogs. In this thesis, descriptors are important artifacts as our proposed method requires them as inputs and generates them as outputs. We will discuss different descriptors defined in the NFV framework in details in Chapter 3. Figure 2.3 shows an overview of NSD.

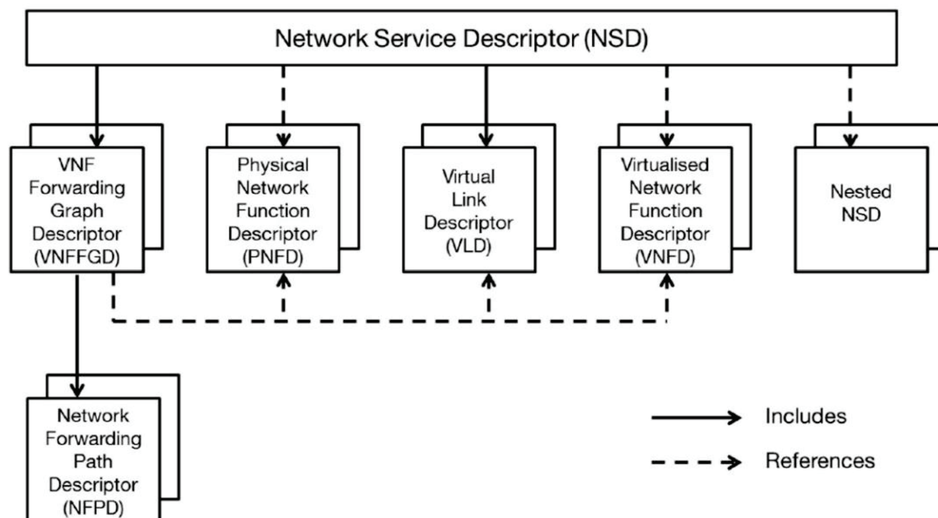

Figure 2.3 - NSD overview (from ETSI GS NFV-IFA 014 [9])

An instance record describes the characteristics of an instantiated entity, e.g. a VNF or NS instance. It includes the instance allocated resources index, operational status, network configurations, e.g. IP address, etc. [8]. In the NFV architecture, records are stored in the NFV instance repository [8], and they are out of our scope.

2.3 Model Driven Development (MDD)

Model Driven Development is a development paradigm that uses models as the primary artifacts of the process [29]. MDD relies on an architecture referred to as Model Driven Architecture (MDA) [16] defined by Object Management Group (OMG) [30]. A model represents an abstraction of a system's structure and/or behavior [31]. This abstraction eases the processing and manipulation of the artifacts as the details are abstracted away. With MDD, models can be transformed into other models for different purposes, such as implementation, validation, generation, etc. An MDD process consists of the models it is manipulating and the transformations it is applying to these models.

2.3.1 Unified Modeling Language (UML)

Unified Modeling Language (UML) [32] is a widely used general-purpose modeling language defined by the OMG. UML is a suit of notations. It enables designers to specify a system using different diagram types [29], e.g. class diagram or activity diagram [33]. UML provides a mechanism called profiling through which we can specialize UML to create a Domain-Specific Modeling Language (DSML). A UML profile is a metamodel that extends the UML metamodel using *stereotypes*, *tagged values* (attributes for *stereotypes*) and *constraints* [29]. There exist many tools supporting UML. Papyrus [34] is a UML tool that provides most of the UML features, including UML diagrams and UML profiling feature. We have used Papyrus in our prototype tool for developing profiles according to our metamodels.

2.3.2 Model Transformation

A model transformation can be Model-to-Model (M2M) or Model-to-Text (M2T). In the former type, one or multiple source models are transformed into one or multiple target models. In the latter, the output is a text string [29]. A model transformation is a mapping function from the source model(s) to the target model(s)/text string [35].

ATLAS Transformation Language (ATL) [36] is a widely used M2M model transformation language. It is a rule-based language and uses OCL [37] expressions and data types to define the algorithms, but it is not limited to OCL. ATL is a hybrid language designed based on both imperative and declarative programming paradigms [29]. An ATL transformation is composed of one or multiple `Rules` and `Helpers`. Each `Rule` generates one or multiple elements in one of the output models from elements in the input models. There are two types of `Rules` including `Matched Rules` and `Lazy Rules`. The former type matches some specified elements in a source model, and from them, generates a group of distinct elements for the target models. `Matched Rules` are invoked declaratively. The latter type has the same functionality, but it is invoked from other `Rules` imperatively. A `Helper` is a method that makes it possible to factorize a piece of code and reuse it. `Helpers` are written using OCL expressions [36].

2.4 Related Work

In the state of the art, different works in different domains are related to this thesis from different aspects. However, to the best of our knowledge, there is no work that addresses the requirement decomposition for the purpose of network service design in the context of NFV.

As discussed in Chapter 1, our work was inspired by [11]. In this work, authors propose a model-driven method for designing highly available applications that satisfy the user require-

ments. The designed applications are composed of Commercial-off-the-shelf (COTS) components in the context of Service Availability Forum (SA Forum) [38] compliant middleware. This method decomposes the user requirements to lower level requirements (referred to as configuration requirements) using an ontology. The method selects the components satisfying the requirements from a catalog. Our approach designs NSs in the NFV context. It requires the selection of VNFs but also the design of other NS constituents, especially the forwarding graphs, which lead to designing the NS traffic flows. This aspect is not considered in [11].

The works in [39, 40] focus on web service composition. In [39], the authors propose a formal meta-framework to compose web services according to functional requirements. It decomposes complex requirements into Boolean combinations of atomic requirements expressed in a certain formalism. It selects the web services that satisfy these atomic requirements. They use available methods for this decomposition and selection. The meta-framework identifies the composition of the selected web services by using satisfiability techniques and reusing the prior composition results. In [40], the authors of [39] extend their meta-framework to take into account the non-functional requirements using formal methods. They analyze users' preferences over the non-functional requirements to find the optimal web service composition. This work does not consider the dependencies and the flows in the composition. This work is applicable for web services but not for the composition of VNFs.

Many works have been done on service composition and decomposition in the context of service-oriented architecture (SOA) [41, 42, 43]. In [41], the authors propose a decomposition-based method to compose services from components according to user requirements. This method takes the QoS of the composed service into account according to its utility. The authors have defined composition structures, by which they compose the services. The method derives

the constraints of the components (for component selection) from the composite service constraint. Then, it computes the utility of the composed service according to the utility of the components. This work considers the flow between the services according to the pre-defined structures.

In [42], the authors propose a method for service composition from atomic services using genetic algorithms. The method uses path decomposition by adopting Case-Based Reasoning and genetic algorithms. It adjusts the execution path and accordingly forms an execution plan to meet the user requirements. The execution paths and plans are correspondents to forwarding graphs at different levels of abstraction. This method designs the composite services to meet the QoS requirements.

In [43], the authors propose an ontology-based method that decomposes IT service processes. They have proposed a structured description of services and service processes that support this method. They have used “server deployment service” as a case study on IT services. It is composed of lower level services including configuration requirements setup, server environment setup, and system configuration setup. The definition of service in this work is different from our definition, and it maps to NFV-MANO processes, to some extent. This work assists service providers to manage their operational processes. This work does not decompose the requirements, nor designs a service or a process.

The works proposed in [44, 45, 46] focus on different aspects of NFV including NSs and service chaining. In [44], authors propose a semantic-based ontology for NSD according to the ETSI NFV standard [8]. They define the relationships between the NSD parameters and construct the ontology accordingly. They have used OWL as the language for constructing the ontology. The purpose of this work is data modeling for VNF management automation and NS generation. However, this work does not propose any method for these activities.

In [45], authors propose an algorithm to provide an efficient placement for a Network Service Chaining (NSC) in the infrastructure. An NSC is composed of multiple NFs, and each NF may have different decompositions. By considering different decompositions, the algorithm maps the NFs to the components of the infrastructure. According to the characteristics of the components, it selects the NF decompositions and realizes an efficient placement. This work assumes the different decompositions are given and it does not decompose the service itself.

In [46], authors have proposed three different architectures for the deployment of IMS in an NFV environment using VNFs. These architectures include typical, merged, and split IMS. Typical IMS complies with the 3GPP standard. Merged IMS combines the IMS blocks into one VNF and deploys one instance for each user. Split IMS decomposes the IMS functionalities into simpler functionalities realized by different VNFs. Furthermore, they propose a management architecture to orchestrate the proposed architectures on top of the cloud infrastructure. We used the merged IMS architecture proposed in this paper to enrich our case study discussed in Chapter 6.

Chapter 3

The Modeling Framework

In this chapter, we will introduce the modeling framework on which our approach for generating network service descriptors is based. This framework consists of a number of metamodels that we propose in addition to the metamodels defined by ETSI NFV. The metamodels we propose are *Network Service Requirement* (*NSReq*), *Network Function Ontology* (*NFO*), *VNF Architecture Descriptor* (*VNFAD*), *VNF Catalog*, *Protocol Stack*, and *Solution Map* (*SM*). The rest of the metamodels are *VNF Descriptor* (*VNFD*), *VNF Package Info*, and *NSD* which ETSI NFV has defined [9, 14, 8, 10]. Our metamodels are based on the UML language [33].

3.1 Network Service Requirements (NSReq)

A tenant expresses his/her requirements for a network service using an *NSReq* model. An *NSReq* model describes a required network service at a high level of abstraction from functional, architectural, and non-functional aspects.

The *NSReq* indicates the functionalities of the required network service. The *NSReq* can constrain these functionalities to be realized only with specific architectures. It also specifies the functionalities that the environment can access through service access points. The *NSReq* can define the required QoS for the network service. All these characteristics can be modeled

using four types of requirement elements defined in the *NSReq* metamodel. These elements are *Functional*, *Architectural*, *Service Access Point*, and *Non-functional Requirements* (*FR*, *AR*, *SAPR*, and *NFR* respectively). The *NSReq* metamodel is depicted in Figure 3.1.

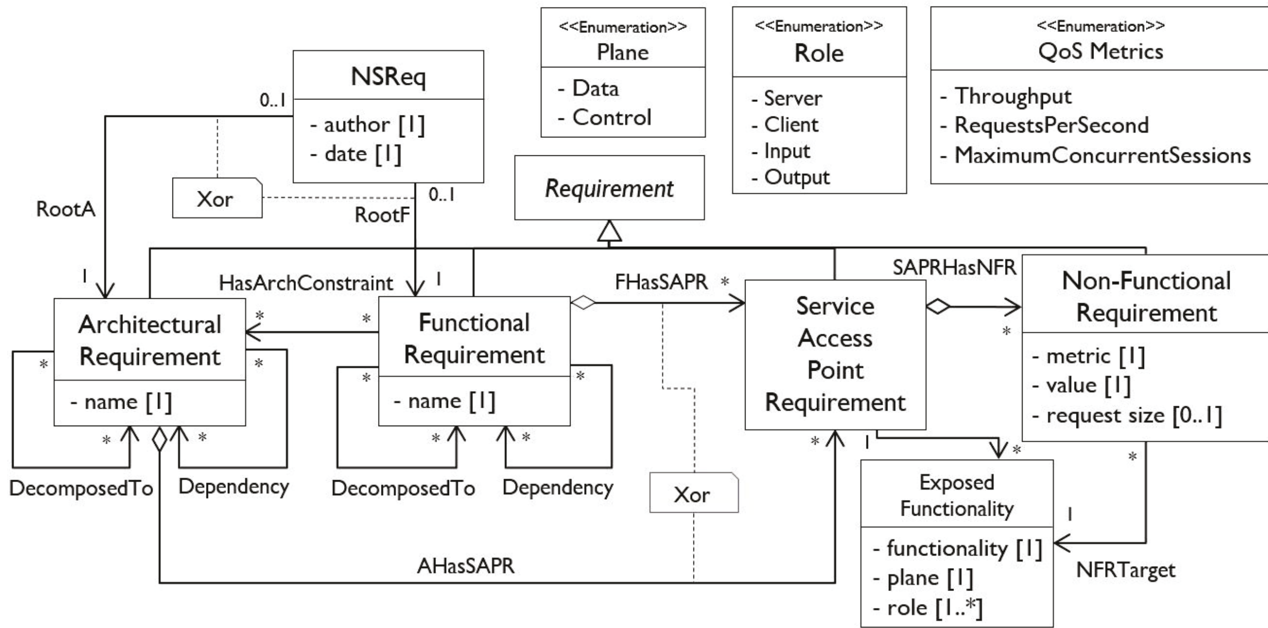

Figure 3.1 - The NSReq metamodel

3.1.1 Functional Requirements (FR)

An *FR* element represents a required functionality in the required network service, specified with a name. A functionality can be decomposed into multiple simpler functionalities. A tenant can require a specific decomposition for a functionality using the *DecomposedTo* association of the *FR*, as shown in the metamodel. Therefore, the *NSReq* structure is hierarchical for the functional requirements. The root of this hierarchy defines the functionality of the required network service at the highest level. This root is indicated by the *RootF* association in Figure 3.1. A functionality can be dependent on other functionalities. It means the dependent functionality requires those other functionalities in order to perform. The tenant can specify such a relation in the required network service using the *Dependency* association of the *FR* element, as shown in the metamodel.

3.1.2 Architectural Requirements (AR)

An *AR* element represents a functional block in an architecture, specified with a name. We refer to these blocks as architectural blocks, and they are realizations of one or multiple functionalities. Generally, an *AR* is a constraint on one or multiple *FR*s. It means only the indicated architectural block should realize those functionalities in the required network service. For example, a tenant can constraint the required voice call functionality to be realized only by IMS architecture. *FR*s are constrained by *AR*s using *HasArchConstraint* association in the metamodel.

The architectural blocks can be composed of multiple simpler architectural blocks. A tenant can require a specific decomposition for an *AR* using the *AR*'s *DecomposedTo* association in the metamodel. The *NSReq* structure is hierarchical for *AR*s as well as the *FR*s. The children of a required functionality inherit the architectural constraints on their parent, i.e. only the children of the constraining *AR* should realize that functionality's children. Architectural blocks may depend on each other in order to function, and it shows the communication between them. The tenant can require such a relation using the *AR'*s *Dependency* association in the metamodel.

There is a special case when the tenant requires an architecture with all its functionalities. In such a case, there should be no required functionalities in the *NSReq*, and the *ARs* exist independent of any functional requirements.

3.1.3 Service Access Point Requirements (SAPR)

A *SAPR* element indicates a required SAP in the required network service. The *SAPR* indicates the functionalities exposed to the environment through the required SAP. The *SAPR* specifies the plane (data/control) and the role(s) (server, client, input, and output) of each function-

ality exposed by the SAP – we will discuss these characteristics of exposed functionalities further in Section 3.2.2. A tenant can specify each exposed functionality associated with a *SAPR* using the *Exposed Functionality* element in the metamodel. The tenant might not have the knowledge for the network service decomposition regarding the exposed functionalities. As a result, these functionalities may or may not exist as an *FR* in the *NSReq*, and they are specified only with a name in the *Exposed Functionality* elements.

3.1.4 Non-functional Requirements (NFR)

An *NFR* element defines the maximum required QoS of a functionality that the environment receives from a SAP. The *NFR* indicates this functionality by an association with an *Exposed Functionality* element, as shown in Figure 3.1. An *NFR* defines the required QoS with a metric and its required value as shown in the metamodel. The metric can be throughput, requests per second (RPS), or maximum concurrent sessions (MCS).

If the metric is RPS, the *NFR* should indicate the total request size using the *Request Size* attribute. Multiplying the request size by the value results in the required throughput. In this case, the *NFR* is composite as it is composed of two QoS requirements (RPS and throughput). A functionality exposed by a SAP cannot associate with more than one *NFR* of each specific metric, e.g. a functionality cannot have two different throughput *NFRs*. It also means a functionality cannot associate with two *NFR*s of RPS and throughput simultaneously, as the *NFR* of RPS metric conveys a throughput requirement as well.

Figure 3.2 depicts an *NSReq* example with the root FR (*FR1*) decomposed into three other FRs (*FR2*, *FR3*, and *FR4*), in which *FR2* and *3* are dependent on *FR4*. A *SAPR* with an *NFR* is defined for each of *FR2* and *FR3*. In this picture, modeling the *SAPR* elements and their relation with *NFR*s are simplified to be easier to follow.

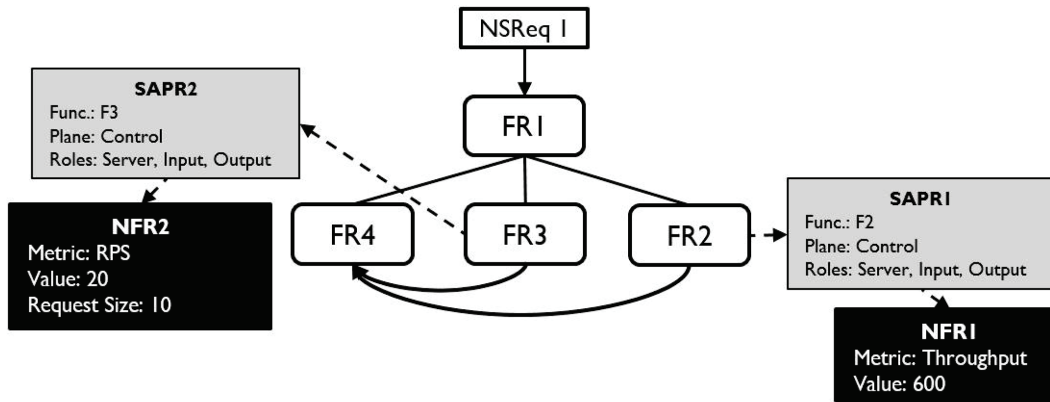

Figure 3.2 – An example of NSReq model

3.2 Network Function Ontology (NFO)

An *NFO* model is a knowledge-base for designing network services. As discussed earlier, the requirements that a tenant provides in an *NSReq* model are at a high level of abstraction. Therefore, there is a gap between these requirements and the required *NSD*. We use the knowledge accumulated in the *NFO* to fill this gap.

An *NFO* model has the information on functionalities, how they decompose to simpler functionalities, and their dependencies. It has the information on what standard architectures and implementations exist to realize these functionalities. The *NFO* describes these architectures' decompositions to lower level blocks, their interfaces, and their dependencies. An expert may develop an *NFO* model manually using the knowledge in the domain of different network services. We will also propose an approach in Section 4.2.7 to automatically enrich the *NFO* to some extent. The *NFO* metamodel is depicted in Figure 3.3. It has three main elements including *Functionality*, *Architectural Block* (*AB*), and *SAP*.

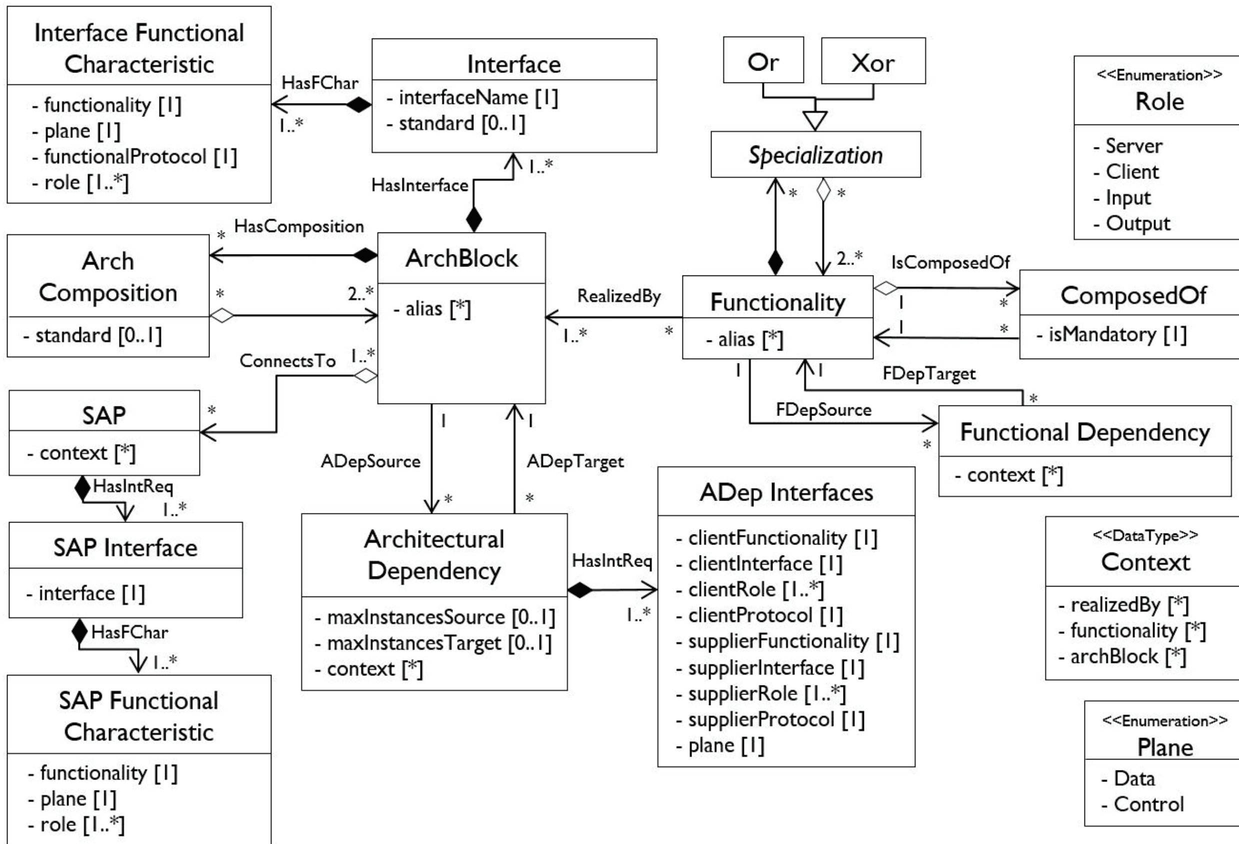

Figure 3.3 - The NF Ontology metamodel

3.2.1 Functionalities

A *Functionality* element corresponds to a functionality that can be part of a network service. It is specified with a name and may have multiple aliases. In an *NFO* model, each functionality is unique, and may have the following relations with other functionalities.

Decomposition: A functionality can decompose to lower level functionalities. Therefore, *NFO* has a hierarchical structure for functionalities. In the decomposition of a functionality, the child functionalities can be mandatory or optional. The mandatory children form the core composition of the functionality. A tenant can customize a functionality's composition by selecting its optional children. In an *NFO* model, we use the *ComposedOf* elements to specify the decomposition of a *Functionality*, as shown in Figure 3.3.

Specialization: Some functionalities are abstract, and there is no implementation for them. Such functionalities can be specialized to more specific functionalities that have been implemented. For instance, messaging functionality is abstract, and it is specialized to immediate messaging and session-based messaging [23]. A tenant might require a general functionality since he/she might not have enough knowledge regarding the functionalities. However, we cannot use general functionalities in designing a network service. Instead of an abstract functionality, we should use one or multiple of its specializing functionalities.

The idea of specialization of a functionality in our *NFO* comes from the feature modeling domain [47]. The specialization can be exclusive or non-exclusive. In an exclusive specialization, only one of the special functionalities should be selected. In an *NFO* model, we use *OR* and *XOR* elements to specialize a *Functionality* (for non-exclusive and exclusive respectively), as shown in Figure 3.3.

Dependency: Functionalities can depend on each other. In a dependency relation, we refer to the dependent functionality as the client and the other one as the supplier. The client and supplier functionalities communicate according to the client-server architecture [20]. The client functionality acts as the client and the supplier functionality acts as the server, i.e. the former sends requests to the latter. The sequence of these dependencies implies the flows in the higher level functionalities, and therefore in the network services. In an *NFO* model, we specify a dependency relation between two *Functionalities* using a *Functional Dependency* element, as shown in Figure 3.3.

Two functionalities may have a dependency relation only in the context of a parent or an ancestor they have in common. I.e. such dependency exists between them only if they are in the decomposition of that parent/ancestor – the ancestors of a functionality are the parents of that functionality's parents and the parents of the functionality's ancestors which recursively reach

the root. We define the context of a *Functional Dependency* in the *NFO* by the *Context* element, as shown in Figure 3.3. For more details regarding the *Context* element refer to Section 3.2.4.

3.2.2 Architectural Blocks (AB)

An *AB* element in the *NFO* corresponds to an architectural block as defined in Section 3.1.2. It is specified with a name, and it may have multiple aliases, similar to a *Functionality*. As discussed earlier, an architectural block realizes one or multiple functionalities. For instance, S-CSCF is an architectural block in the IMS architecture that realizes the registration and session setup functionalities [13]. We use the *RealizedBy* association to define this relation between the *ABs* and *Functionalities* in the *NFO*, as shown in Figure 3.3. An *AB* may have the following relations and elements.

Decomposition: An architectural block can be decomposed into simpler architectural blocks. Unlike a functionality, an architectural block can have alternative decompositions, some of which can be based on a standard or just an implementation. For instance, an IMS architecture can be decomposed according to the 3GPP standard [13]. We specify a decomposition of an *AB* by an *ArchComposition* element in the *NFO*, as shown in Figure 3.3.

Interface: An architectural block communicates with other architectural blocks through its interfaces. An interface exposes one or multiple functionalities realized by the architectural block on specific planes (data/control) to the outside world. A functionality may perform on multiple planes. Different fractions of the functionality related to different planes can be exposed through different interfaces. As an architectural block realizes a functionality, it also realizes the children of that functionality. Therefore, an exposed functionality through an interface can be either the functionality realized by the architectural block or one of its children. Each

exposed functionality has specific roles (input, output, server, and client) while communicating through a specific interface.

Input and output roles are defined in terms of the direction of the packet flows, and they are complementary. These two roles can be defined on both data and control planes, as there can be packet flows associated with both planes. Server and client roles are according to the client-server architecture [20]. The exposed functionality which sends requests has the client role, and the one that receives them has the server role. These two roles are also complementary, and they are defined only on the control plane. Each exposed functionality uses a specific protocol for communication through the interface.

As an example, Gm is one of the interfaces of P-CSCF architectural block in IMS architecture [13]. It exposes registration and session setup functionalities on the control plane and uses SIP protocol. User equipment sends requests to and receives responses from this interface. Therefore, this interface has the roles of server, input, and output for both functionalities. We specify each interface of an *AB* by the *Interface* element, and each exposed functionality by the *Interface Functional Characteristic* element in the *NFO*, as shown in Figure 3.3.

Dependency: Two architectural blocks may communicate differently in different contexts, i.e. in different decompositions and/or while realizing different functionalities. In each communication between two architectural blocks, an exposed functionality from each of their interfaces is involved. Each involved functionality has specific roles and uses a specific protocol in such communication. These characteristics are a subset of the characteristics exposed by the interface. The communication is also on a specific plane, which both involved interfaces should support. We define the communication between two *ABs* in the *NFO* by the *Architectural Dependency* element. The characteristics of the functionalities involved in the communication are indicated using *ADep Interfaces* element, as shown in Figure 3.3.

We define the context in which a dependency relation exists by using the *Context* element. A *Context* element may define one or many parents/ancestors that both *AB*s of a dependency relation have in common. I.e. those *ABs* have that communication only if they are in the decomposition of the indicated parents/ancestors. The *Context* element may also define *RealizedBy* associations related to one or both *AB*s. It implies that those *AB*s have such communication only if they realize the indicated *Functionalities*. For more details regarding the *Context* element refer to Section 3.2.4.

3.2.3 Service Access Points (SAP)

A *SAP* element indicates an access point of a whole architecture. The environment communicates with an architecture (or a network service) through such access points. An access point (service access point) exposes one or some of the architectural blocks' interfaces to the environment. The access point may expose all or a subset of these interfaces' characteristics. Therefore, the environment can access the exposed functionalities through such interfaces.

For instance, the Gm interface of P-CSCF block in IMS architecture is exposed to the environment. The user equipment can communicate with P-CSCF through the access point associated with the Gm interface for registration and session setup [13]. We indicate the interfaces and the subset of their characteristics exposed by a *SAP* using the *SAP Interface* and *SAP Functional Characteristic* elements, respectively, as shown in Figure 3.3.

A *SAP* may have specific contexts, similar to *Architectural Dependency* as discussed earlier. For instance, a *SAP* exposing an interface of an *AB* may exist only if the *AB* is in a specific decomposition. We define the context of a *SAP* using the *Context* element. It defines the parent/ancestor(s) of the *AB*s associated with the *SAP*. For more details regarding the *Context* element refer to Section 3.2.4.

3.2.4 Context

As discussed in the previous subsections, the *Context* element defines the context in which the *Functional Dependency, Architectural Dependency,* and *SAP* elements exist. When the *Context* is used for a *Functional Dependency,* only its '*functionality*' attribute is applicable. When it is used for an *Architectural Dependency,* its '*realizedBy*' and '*archBlock*' attributes are applicable. When it is used for a *SAP,* only its '*archBlock*' attribute is applicable.

The way we have defined the *Context* element is that there is a logical AND relationship among its attributes. It means the context is valid only if all of the indicated elements by the attributes exist. Also, there is a logical OR relationship among all the *Context* elements defined for an element, i.e. that element exists if at least one of the contexts is valid. This way, defining the context for an element is more flexible.

Figure 3.4 shows an example of an *NFO* model. In this example, we have omitted *Interface*, *ADep Interfaces*, and *SAP Interface* elements. We are showing each *SAP* using a black dot connected to an *Architectural Block*. The functionalities that each *SAP* exposes from its *Architectural Block* is written on the edge that connects the *SAP*.

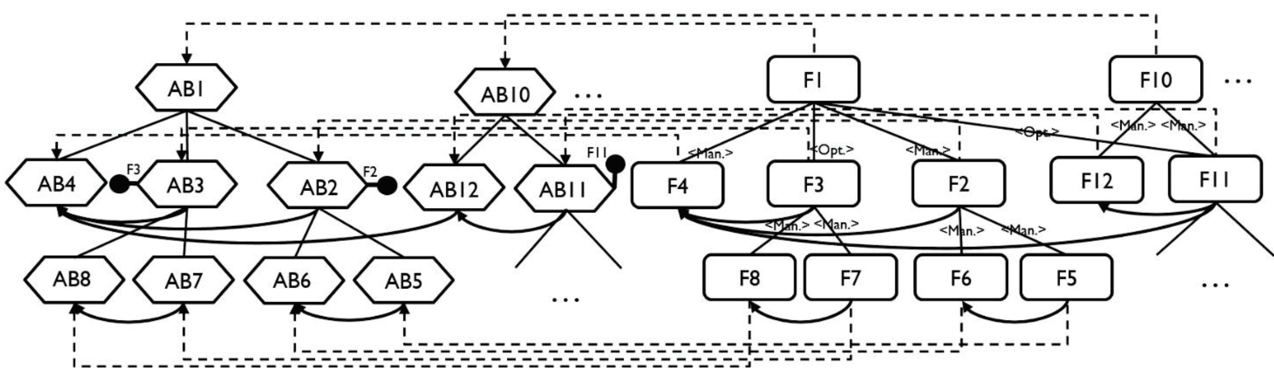

Figure 3.4 - An example of NFO model

3.3 VNF information elements

A VNF has two main information elements used in this work including *VNFD* and *VNFAD*. The *VNFD* describes the VNF from the virtualization perspective, i.e. describing its constituents and the requirements for its deployment [14, 10]. In order to design network services out of VNFs, we need the information of the VNFs application aspect, e.g. their functionalities and application interfaces. Such information does not exist in the *VNFD*, and it is out of ETSI NFV's scope. We have proposed a new information element for VNFs which include such information, i.e. *VNFAD*. These two information elements along with other elements describing a VNF are entailed by a *VNF Package Info* model that is defined and standardized by ETSI NFV [14, 10].

3.3.1 VNF Descriptor (VNFD)

A *VNFD* model is the deployment template of a VNF, and it describes the virtualization characteristics of the VNF and its constituents. The VNF vendors provide the *VNFD*s, and the NFVO uses them in order to orchestrate and manage the VNFs in the NFV framework. The characteristics that a *VNFD* defines are the requirements that the NFVI should provide in order to deploy the VNF. ETSI NFV has standardized this model in [14].

A *VNFD* model is composed of a descriptor for each of the VNF's elements, e.g. VNFC, VnfVl, and VnfExtCp. Their descriptors are called *Virtual Deployment Unit* (*Vdu*), *VnfVl Descriptor* (*VnfVLD*), and *VnfExtCp Descriptor* (*VnfExtCpd*) respectively. In addition, the *VNFD* model describes the VNF's different compositions and non-functional characteristics using deployment flavor elements. Figure 3.5 depicts a portion of the *VNFD* metamodel which is in our scope. For the complete metamodel refer to [14].

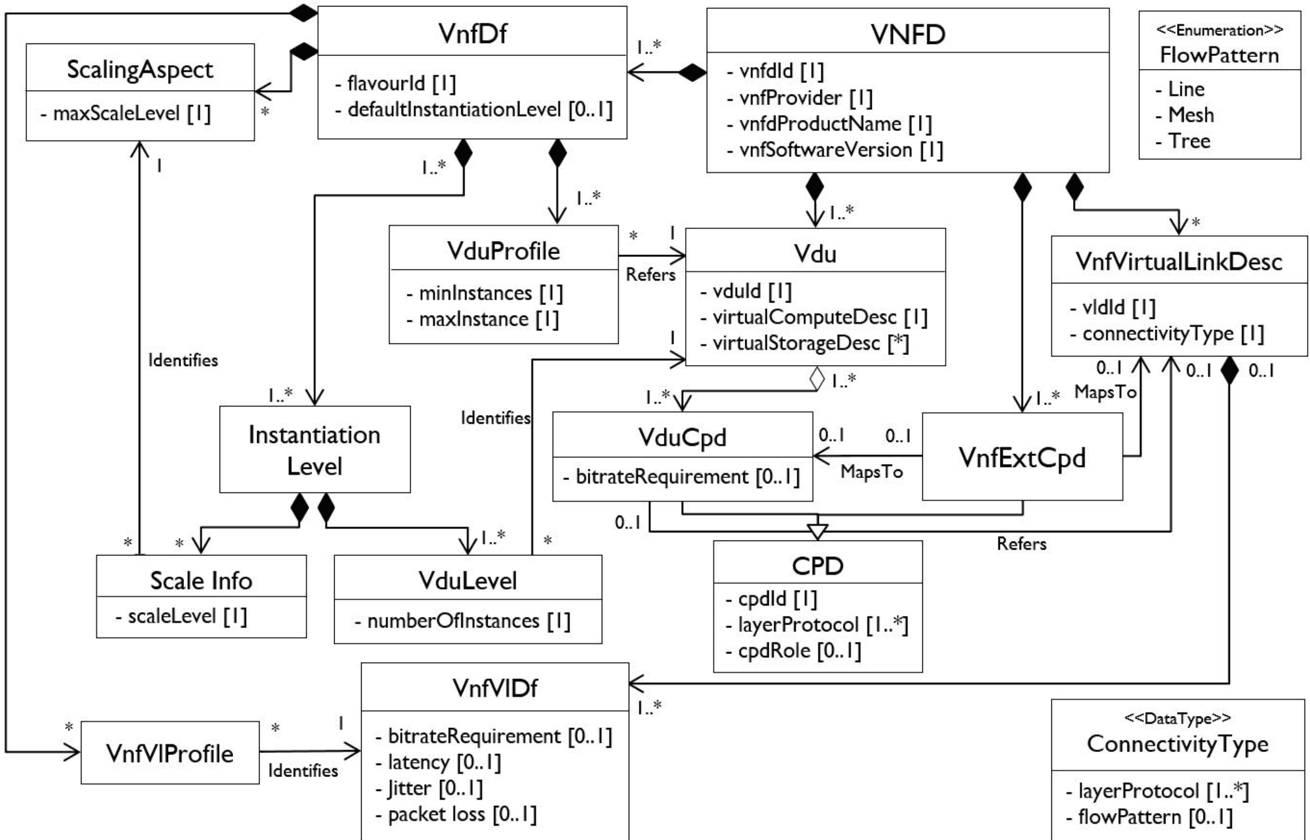

Figure 3.5 - The VNFD metamodel

3.3.1.1 *Vdu*

A *Vdu* element describes the deployment requirements of a VNFC, e.g. required CPU, RAM, and storage. It describes the VnfcCps using *VduCpd* elements, as shown in Figure 3.5. A *VduCpd* specifies the maximum bitrate that the connection point can provide. It defines the layer protocols that the connection point uses for communication, e.g. IPv4, IPv6, Ethernet, etc. It also describes the role of the connection point in the flow pattern of the VnfVl it is involved with. For instance, it specifies whether a connection point's role is root or leaf in an E-Tree flow pattern.

3.3.1.2 *VnfVLD*

A VnfVLD element is the deployment template of a VnfVl of the VNF. It indicates the flow pattern of the virtual link, i.e. E-Line/E-LAN/E-Tree [28, 10], and the layer protocol it uses for

communication, e.g. IPv4, IPv6, Ethernet, etc. The *VnfVLD* uses the *ConnectivityType* element to provide this information, as shown in Figure 3.5.

By using deployment flavor elements (*VnfVlDf*), the *VnfVLD* describes different levels of QoS that the virtual link provides. These deployment flavors define the required bitrate, latency, jitter, and packet loss ratio of the VnfVl. The QoS of the internal virtual links is out of our scope, as we are interested in the QoS that the whole VNF exposed to the environment.

3.3.1.3 *VnfExtCpd*

A *VnfExtCpd* element describes a VnfExtCp. It indicates the connection point's connectivity to an internal connection point or an internal virtual link. This connectivity is to a virtual link when the external connection point should connect to multiple internal connection points. The *VnfExtCpd* defines the layer protocols that the connection point uses and its role in the flow pattern, similar to the *VduCpd*, as discussed in Section 3.3.1.1.

3.3.1.4 *VnfDf*

A *VnfDf* element is a deployment flavor that describes how the VNF is composed of VNFCs and the VNF's non-functional characteristics. A VNF may have different deployment flavors. Each flavor indicates a group of VNFCs that together realize the VNF. Therefore, different flavors may define different sets of functionalities for the VNF. A *VnfDf* indicates the involvement of a VNFC using a *Vdu Profile* element, as shown in Figure 3.5. In a deployment flavor, each VNFC can have a number of instances within the range defined by its profile.

Since each VNFC in a deployment flavor has a range of the number of instances, each deployment flavor defines a capacity range for the VNF. A deployment flavor defines multiple instantiation levels in this capacity range. Each level defines the exact number of instances for each VNFC involved in the flavor. A VNF is instantiated according to one of its instantiation

levels of its selected deployment flavor. The *VnfDf* defines each instantiation level using an *Instantiation Level* element, as shown in Figure 3.5. From each instantiation level, the VNF can be scaled according to different scale levels.

A deployment flavor defines other VNF's characteristics including affinity/anti-affinity rules for the VNF's elements, QoS of the internal virtual links, scaling aspects, etc. These characteristics are out of our scope.

3.3.2 VNF Architecture Descriptor (VNFAD)

A *VNFAD* describes the application aspect of a VNF – in this thesis it only provides the portion required for the NS design. It specifies the VNF's functionalities by specifying the architectural blocks that the VNF is implemented based on. It specifies the VNFCs' functionalities, the VNF's application interfaces, and the QoS provided through these interfaces. It defines different flows inside the VNF related to different functionalities. Figure 3.6 depicts the *VNFAD* metamodel that we propose. Among the *VNFAD*'s elements, the *VNF Interface* and *Flow Transformation* require further discussions.

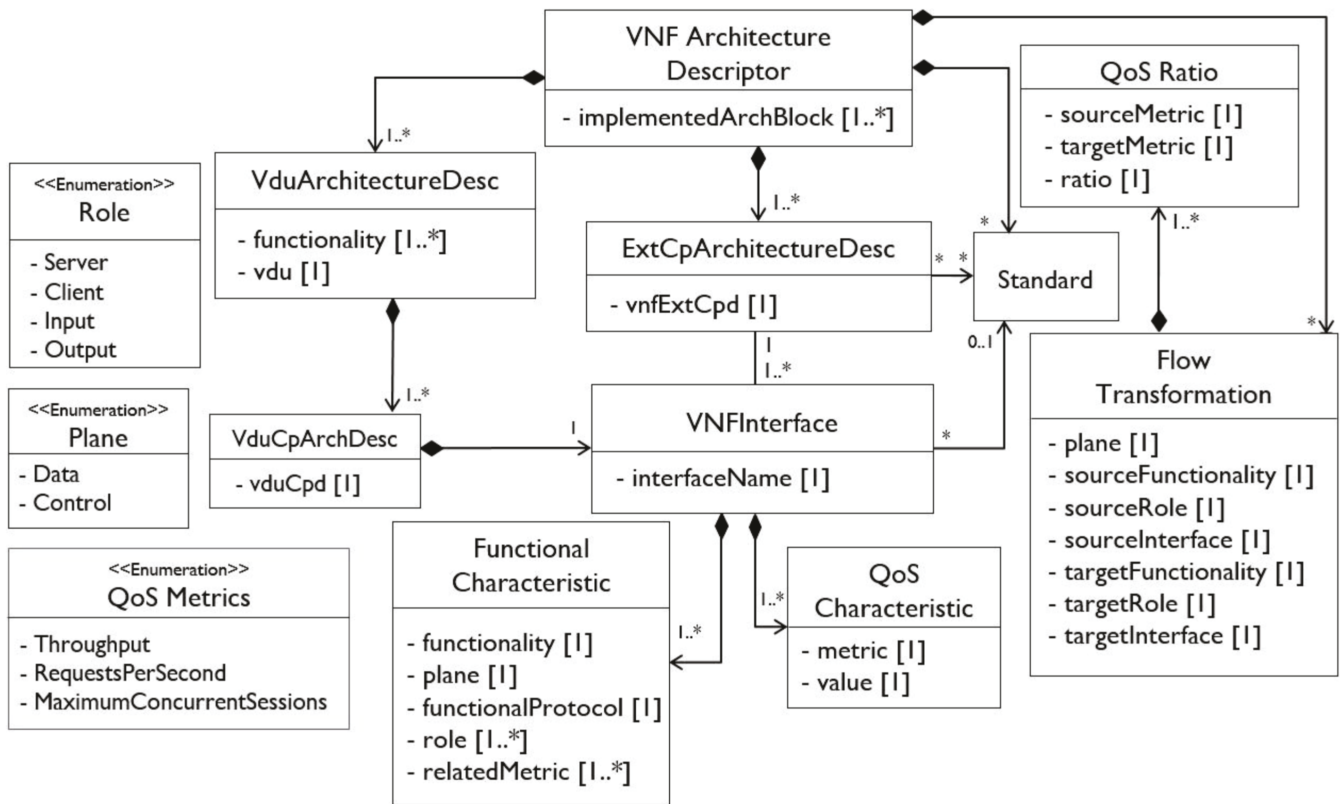

Figure 3.6 - The VNFAD metamodel

3.3.2.1 *VNF Interface*

A *VNF Interface* element defines an application interface of the VNF exposed to the environment. VNFCs expose these interfaces through their VnfcCps which are exposed through VnfExtCps. A VNF's interface is the realization of an interface of the architectural block that the VNF implements. Similar to an architectural block's interface, a VNF's interface exposes one or multiple functionalities (or their children) of the VNFC on specific planes. The exposed functionalities have specific roles and use specific protocols in their communication through the interface. A *VNF interface* specifies each of its exposed functionalities and its characteristics using a *Functional Characteristic* element, as shown in Figure 3.6.

We make the assumption that the VNF capacity is compartmentalized in the sense that each VNF interface is associated with a portion of the VNF capacity. Each capacity portion is dedicated to the functionalities exposed by the associated interface. These functionalities share this portion of the VNF capacity, while exposed functionalities of other interfaces cannot access it. The capacity portion associated with each interface results in a specific QoS exposed form that interface.

We characterize this QoS by a value and a metric attribute. The metric can be throughput, RPS, and MCS, same as the metrics for NFRs, as discussed in Section 3.1.4. A metric of an interface might not be appropriate for all its exposed functionalities. For instance, electronic mail functionality is not session based, and MCS metric is not appropriate to describe its QoS. The '*relatedMetric*' attribute in the *Functional Characteristic* element indicates the appropriate metrics for the specified functionality. We define the QoS related to a *VNF Interface* element by *QoS Characteristic* elements, as shown in Figure 3.6.

3.3.2.2 *Flow Transformation*

A *Flow Transformation* element defines a flow inside the VNF. We define such flow by the source and the target interfaces, and the subset of their characteristics related to the flow (a plane, functionality, and role for each interface).

The other characteristic of a flow is the transformation of its QoS from the source interface to the target interface. We define this QoS transformation by defining the source and target metrics and the ratio of the transformation. For instance, a VNF receives a flow at interface-1 with 1 request per second. The VNF sends out 10 units of throughput from its interface-2 per each request. Therefore, the QoS transformation for such flow is RPS → Throughput with a ratio of 10. The metric of an interface in the QoS transformation can be throughput if it has a role of input or output in the flow. Similarly, for RPS and MCS metrics the interface should

have a role of server or client. We define the QoS transformations of a flow by the *QoS Ratio* elements, as shown in Figure 3.6.

3.3.3 VNF Package Info

A *VNF Package Info* model contains the VNF information elements that the VNF vendor has provided. These elements include *VNFD*, *VNFAD*, Software Image Information, Artifact Information, and other elements. Except for the *VNFD* and *VNFAD*, the rest of these elements are out of our scope. After on-boarding a VNF, the NFVO creates its corresponding *VNF Package Info* and stores it in the *VNF Catalog*. Figure 3.7 depicts the *VNF Package Info* metamodel.

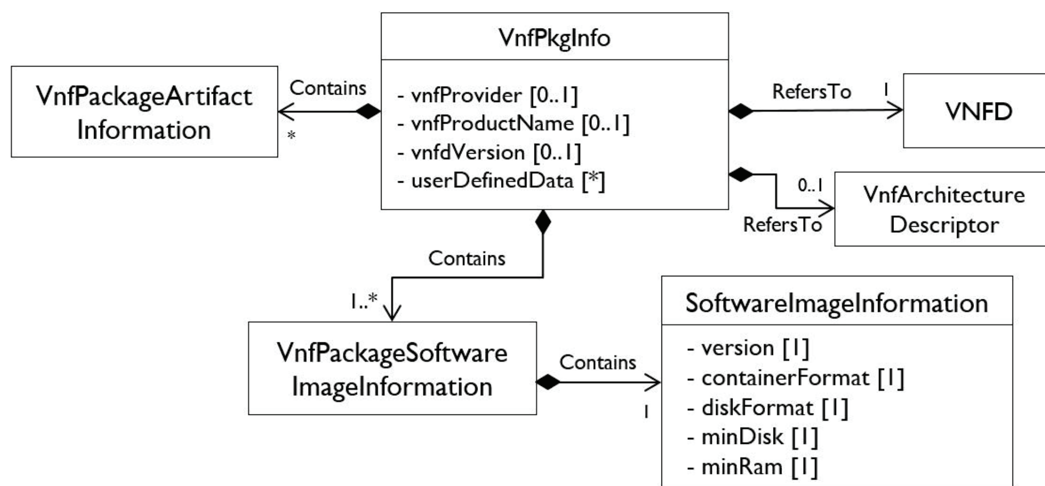

Figure 3.7 - The VNF Package Info metamodel

VNFAD is our proposed information element, and it is not considered in the ETSI NFV's standard. Therefore, *VNF Package Info* element, which is in the standard, does not reference the *VNFAD*. We have used the *User Defined Data* attribute in the *VNF Package Info* element, which is a key-value pair, to keep this reference, as shown in Figure 3.7.

3.4 VNF Catalog

The *VNF Catalog* is a collection of the *VNF Package Info* models of the on-boarded VNFs. For generating *NSD*s, we can access the *VNFDs* and *VNFAD*s through the *VNF Package Info* models in the *VNF Catalog*. Figure 3.8 depicts the *VNF Catalog* metamodel.

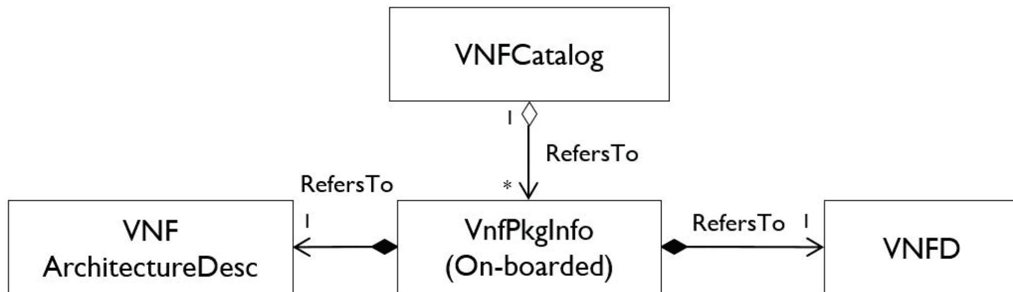

Figure 3.8 – The VNF Catalog metamodel

3.5 Protocol Stack

The *Protocol Stack* model shows which protocol in any layer of TCP/IP protocol stack [48] serves other protocols in the higher layer. We use this information to check whether the protocols of different VNFs are compatible for communication. An expert should create and update this model manually. We have proposed the metamodel shown in Figure 3.9 for the *Protocol Stack*.

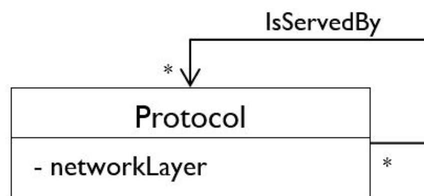

Figure 3.9 – The Protocol Stack metamodel

3.6 Solution Map (SM)

A *Solution Map* (*SM*) model captures the information processed throughout the *NSD* generation process. It captures the information from the input models, i.e. *NSReq*, *NFO*, and *VNF*

Catalog, and their relations. As a result, an important portion of the *SM* metamodel is a combination of the aforementioned input metamodels.

SM model, also, captures the information on the forwarding graphs designed in the *NSD* generation process. These forwarding graphs are *Functional*, *Architectural*, and *Pre-VNF Forwarding Graphs* (*FFG*, *AFG*, and *Pre-VNFFG* respectively), and we will discuss them further in Chapter 4. Figure 3.10 depicts the *SM* metamodel, in which the elements from the *NSReq* model are in cream, from the *NFO* are in blue, from the *VNF Catalog* are in orange, and forwarding graph elements are in green. In addition, the *SM* model contains other elements related to the *NSD* refinement process – discussed in Chapter 5 – which are shown in gray.

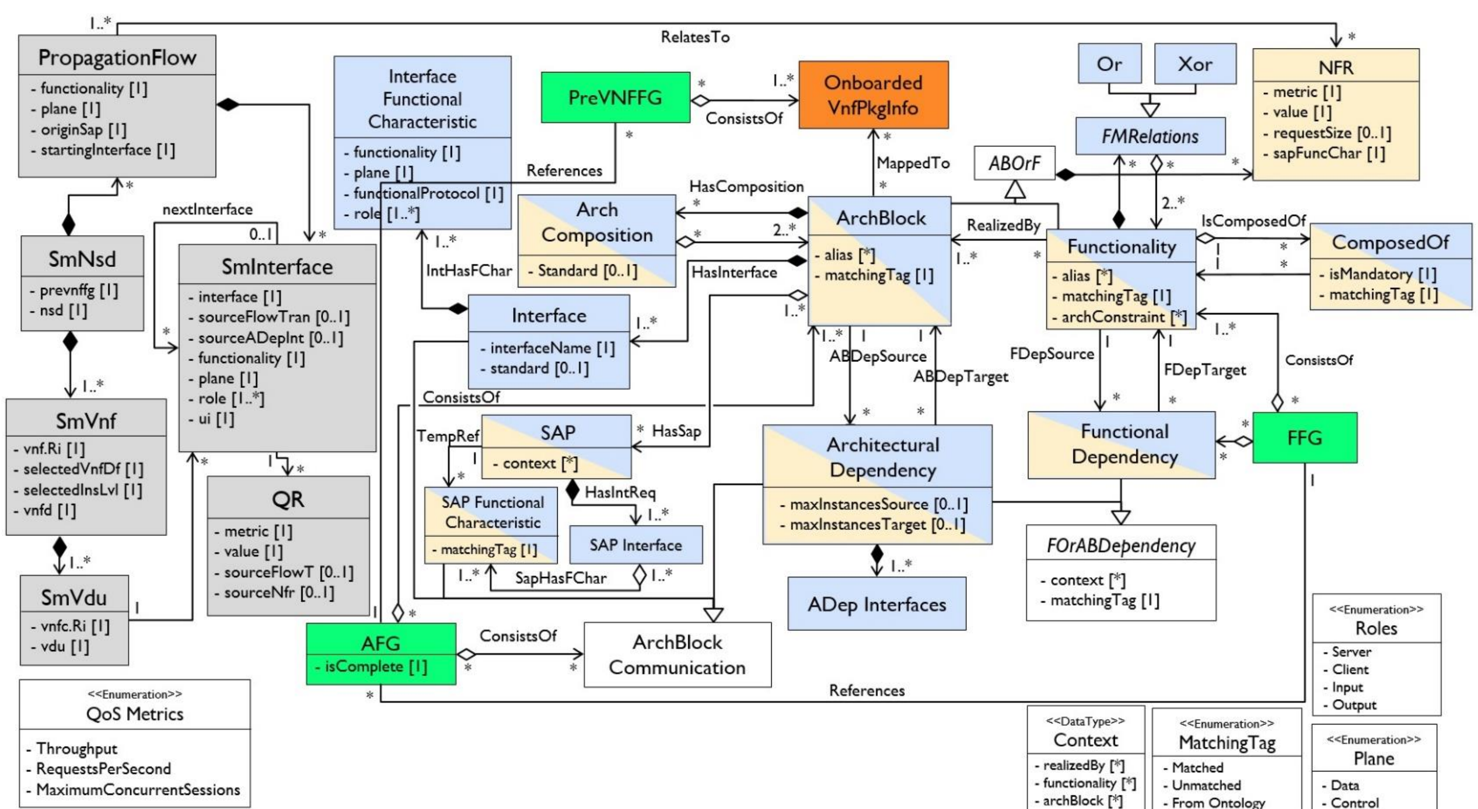

Figure 3.10 - The Solution Map metamodel

3.7 Network Service Descriptor (NSD)

An *NSD* model is the deployment template of a network service. It describes the virtualization aspect of the network service and its constituents. The characteristics that the *NSD* defines are requirements that the infrastructure should provide for the network service, e.g. the VMs' capacity and the virtual networks QoS.

An *NSD* consists of the descriptor of the network service elements, including *VNFDs*, *NsVL Descriptors* (*NsVLD*), *SAP Descriptor* (*SAPD*), and *VNFFG Descriptor* (*VNFFGD*). In addition, the *NSD* describes the composition and non-functional characteristics of the network service using deployment flavor elements. Figure 3.11 shows a portion of the *NSD* metamodel which is in our scope. For the complete metamodel refer to [9].

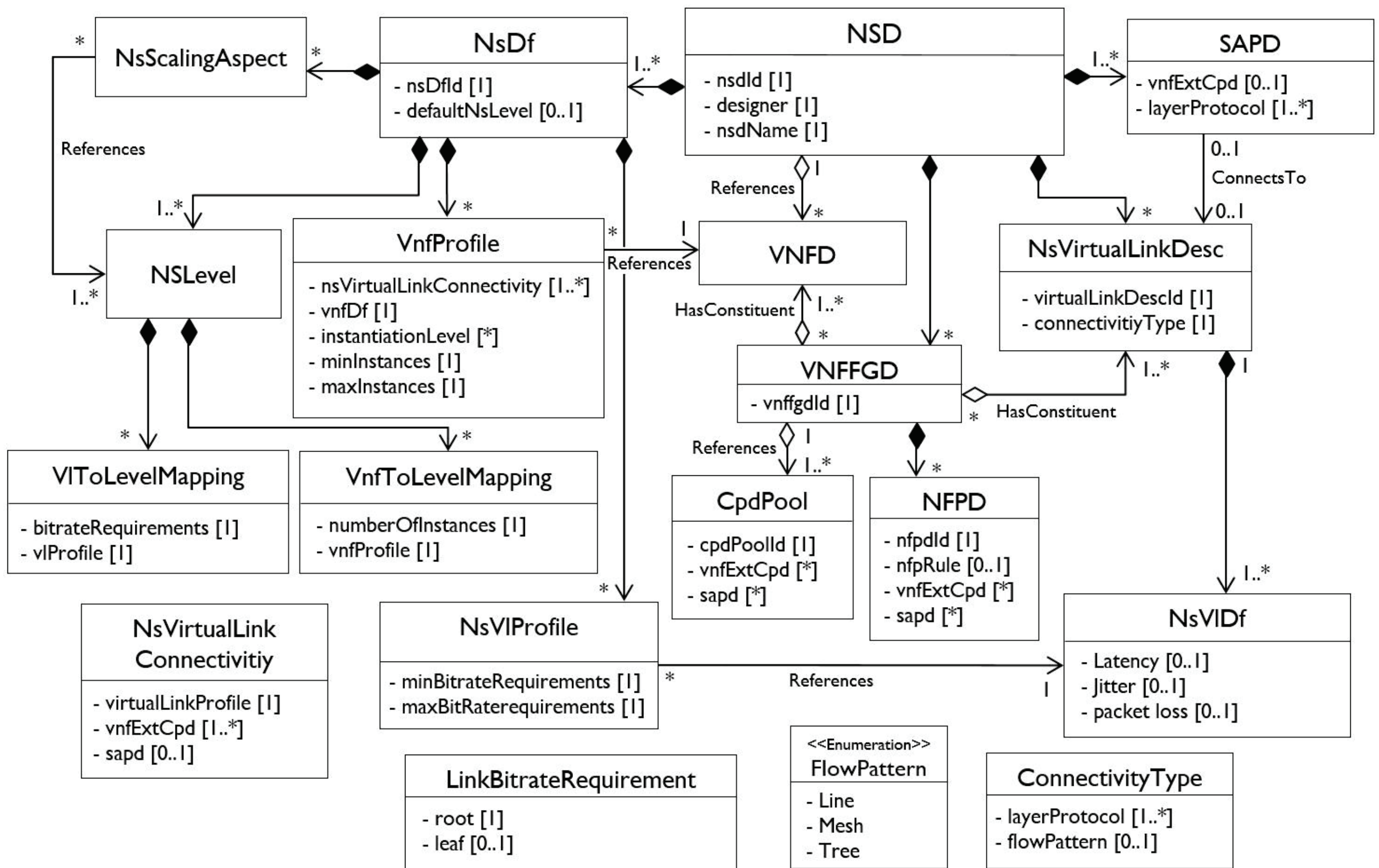

Figure 3.11 - The NSD metamodel

3.7.1 NsVLD

An *NsVLD* element is the deployment template of an NsVl. It defines the flow pattern of the virtual link (*E-Line/E-LAN/E-Tree*) and the layer protocol required for its communication, e.g. IPv4, IPv6, Ethernet, etc. The *NsVLD* uses a *Connectivity Type* element to define these characteristics, as shown in Figure 3.11. The connectivity of a virtual link to VNFs is defined by the *VNF Profile* element.

The *NsVLD* defines the QoS characteristics of the virtual link including latency, jitter and packet loss ratio. A virtual link may have different levels of QoS to be used in different cases. The *NsVLD* defines each QoS level using an *NsVL Deployment Flavor* (*NsVlDf*) element, as shown in Figure 3.11. The aforementioned QoS characteristics of virtual links are out of our scope. The throughput characteristic of a virtual link is described by the *NsVL Profile* element.

3.7.2 SAPD

A *SAPD* element describes the virtualization aspect of a SAP in the network service. It indicates the layer protocol required for its communication, e.g. IPv4, IPv6, Ethernet, etc., and its connectivity to VNFs in the network service. As discussed earlier, a SAP connects to VNFs through their external connection points. Connectivity of a SAP to a single connection point is direct, and to multiple connection points is through an NsVl. The *SAPD* element defines this connectivity by referencing the corresponding *VnfExtCpd* or *NsVLD*, as shown in Figure 3.11.

3.7.3 VNFFGD

A *VNFFGD* element defines a VNFFG in the network service by indicating the VNFFG's constituents. As discussed earlier, a VNFFG is composed of a group of NFPs, and the VNFs, NsVls, and SAPs involved in those NFPs. The *VNFFGD* references the descriptors of the constituent VNFs and NsVls directly. It references the descriptors of the VnfExtCps and SAPs

forming the NFPs using *CpdPool* elements, one *CpdPool* per VNF and one for all the SAPs, as shown in Figure 3.11.

The *VNFFGD* defines each NFP by defining an *NFP Descriptor* element (*NFPD*). An *NFPD* indicates the constituent connection points of the NFP by referencing their descriptors. It also defines the source and destination IP addresses and port ranges using '*nfpRule*' attribute, as shown in Figure 3.11. Assigning the IP addresses and the port ranges is out of our scope since it is a matter of network service configuration.

3.7.4 NsDf

An *NsDf* element is a deployment flavor that specifies a composition, capacity, scalability, and affinity/anti-affinity rules for the network service.

Composition: An *NsDf* indicates a group of VNFs referenced by the *NSD* to realize the network service. Different flavors in a network service may indicate different sets of VNFs, and therefore different functionalities for the network service. An *NsDf* indicates each VNF by a *VNF Profile* element. Different sets of VNFs may require different connectivity. Therefore, the *NsDf* indicates a group of NsVls for the VNF's connectivity. These virtual links should be from the ones that *NSD* references. The *NsDf* indicates each NsVl using an *NsVl Profile* element, as shown in Figure 3.11. The connectivity between the VNFs and virtual links are specified by the *VNF Profile* elements using the *NS Virtual Link Connectivity* elements, as shown in Figure 3.11.

Capacity: Each deployment flavor indicates a capacity range for the network service by defining a capacity range for the VNFs and NsVls in their profiles. Each *VNF Profile* specifies a deployment flavor, an instantiation level, and a range for the number of instances for the VNF. Each *NsVL Profile* specifies a deployment flavor and a bitrate requirement range for the root

and the leaf of the virtual link. The definition for the root and the leaf bitrate requirements for virtual links with different flow patterns are different [10]:

- **E-Line:** The root bitrate requirement is equal to the bitrate of the line. The leaf bitrate requirement is not applicable, as *E-Line* has no leaf connection.
- **E-LAN:** The root bitrate requirement is equal to the aggregate capacity of the LAN. The standards today do not support multiple bitrate requirements for the leaf connections. Therefore, the leaf bitrate requirement is equal to the maximum bitrate among all of the virtual link's connections.
- **E-Tree:** The root bitrate requirement is equal to the virtual link's root bitrate. The leaf bitrate requirement is equal to the maximum bitrate among the virtual link's leaves (for the same reason as the *E-LAN* leaf bitrate requirement).

The *NsDf* defines multiple capacity levels in the capacity range defined for the network service. Each level specifies an exact number of instances for each VNF and the exact bitrate requirement for each NsVl. These values should be in the ranges defined by the profiles. The network service is instantiated according to one of these levels, and it can be scaled from one level to another. An *NsDf* defines each level using an *NS Level* element. The *NS Level* specifies the VNFs' and NsVls' capacities using *VnfToLevelMapping* and *VirtualLinkToLevelMapping* elements respectively, as shown in Figure 3.11.

The network service's scalability and the affinity/anti-affinity rules defined by the deployment flavor are out of our scope.

Chapter 4

Derivation of Network Service Descriptor from Functional and Architectural Requirements

In this chapter, we present our approach for the generation of *NSD* from *NSReq* taking into account the functional and architectural requirements only. The refinement of this approach to take into account the non-functional requirements is described in the next chapter. Our approach generates multiple *NSD*s which we refer to as generic *NSD*s as they are not tailored to specific QoS. In addition, our approach enriches the *NFO* based on new information obtained from the *NSReq* or the *VNF Catalog*.

4.1 Overall approach

As discussed in Section 3.1, the *NSReq* model that a tenant provides generally represents high level requirements. Therefore, there is a gap between these requirements and the targeted *NSD*. To fill this gap, we decompose the *NSReq* recursively with the help of the *NFO*. To do so, we map the *NSReq*'s requirements to the *NFO* elements. Then, we add the decomposition of the matched elements from the *NFO* to the *SM*. This way, we decompose these requirements to lower level requirements. According to these low level requirements, we are able to select the

set of VNFs from the *VNF Catalog*. Different combinations of VNFs from this set can realize the required network service. We capture each of these combinations and their VNFs' connectivity. Each combination forms a forwarding graph which is the main element of a network service. From each forwarding graph, we generate a generic *NSD* element. Finally, we may enrich the *NFO* according to new information obtained from the *NSReq* model.

In this document, we use the term "mapping" for a specific operation between models or elements of the models. By mapping two elements we mean matching their attributes' values in order to realize if the two elements match. By mapping two models we mean mapping a group or all of their elements. The result of mapping two models can be the decomposition or selection of some elements depending on the models. Figure 4.1 shows the overall picture of the approach.

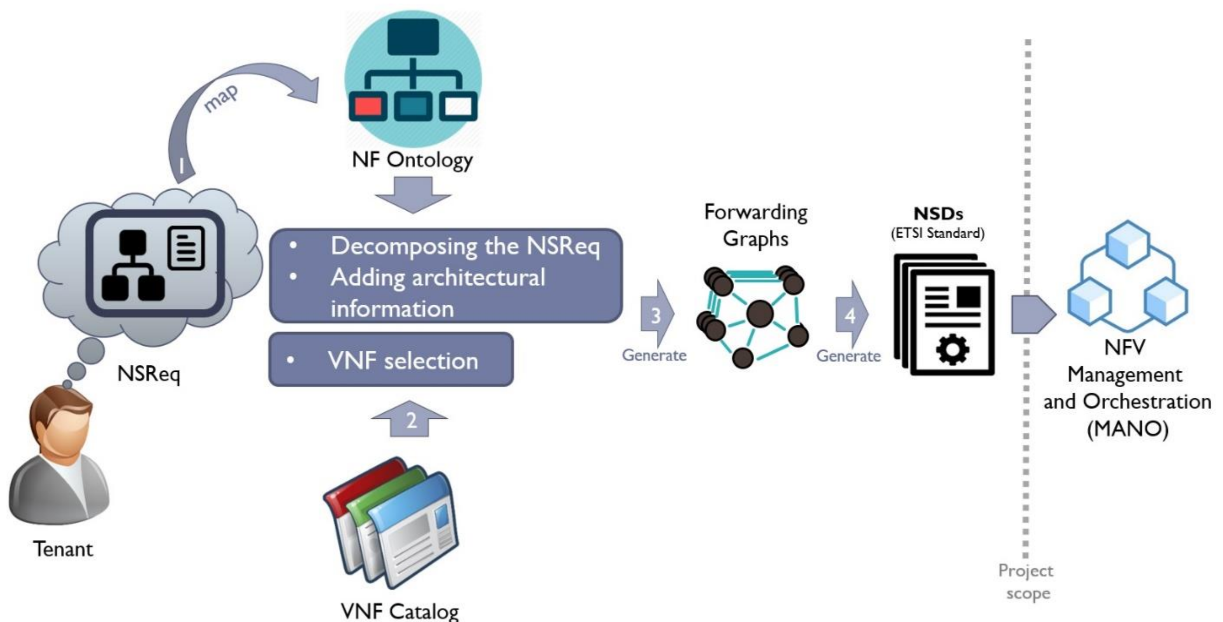

Figure 4.1 - The overall picture of the NSD generation process

The input of this approach is the *NSReq*, *NFO*, *VNF Catalog*, and *Protocol Stack* models. Our approach for *NSD* generation from *NSReq* consists of six steps:

- Step 1 – Initialization of the *SM* model

- Step 2 – Decomposing the *SM* model
- Step 3 – Selecting the *VNF*s
- Step 4 – Generating the forwarding graphs
- Step 5 – Generating generic *NSD*s
- Step 6 – Updating the *NFO*

Figure 4.2 shows the input/output models of each of these steps.

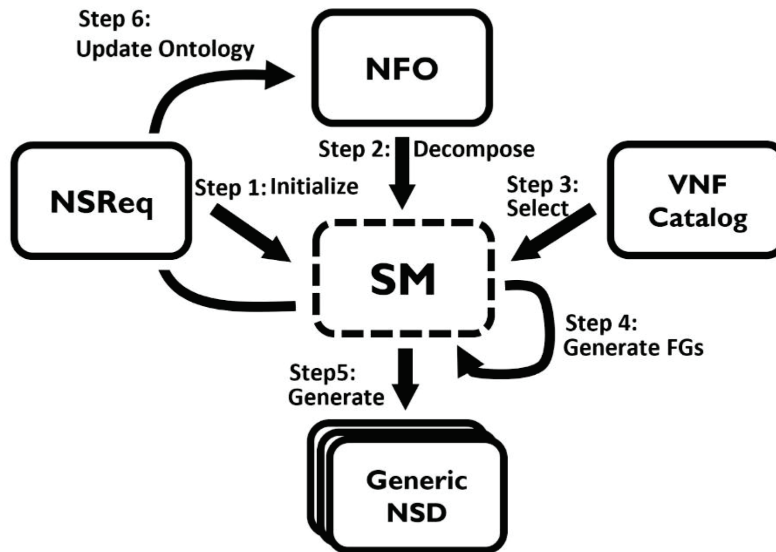

Figure 4.2 - The steps of the NSD generation process and the information flow

4.2 Steps of the Approach

In this section, we will discuss each step of the approach in details.

4.2.1 Step 1 – Initialization of the SM model

Throughout the whole *NSD* Generation process, we use an *SM* model to capture all the information needed from each input model. It simplifies the manipulation of the required information and therefore simplifies the method. The *NSReq* model is the starting point of the process. In the first step, we initialize an *SM* model from the *NSReq*. For this, we transform all the *NSReq* elements into their corresponding *SM* elements.

Functional and Architectural Requirements: In the next step (Step 2), we map the functional and architectural requirements in the *NSReq* to the functionalities and architectural blocks in the *NFO*, respectively. To simplify this mapping, in this first step, we transform the *FR and AR* elements in the *NSReq* into the *Functionality* and *AB* elements in the *SM*, respectively. For each *FR*, we transform its decomposition and dependency relations into *ComposedOf* and *Functional Dependency* associations in the *SM,* respectively. For each *AR*, we transform its dependency relations into *Architectural Dependency* associations in the *SM*. For the whole decomposition of each *AR*, we create an *Arch Composition* element in the *SM*. Then, we associate it with the *AB*s in the *SM* that correspond to that *AR* and its children.

Service Access Point Requirements: In the next step (Step 2), we map the service access point requirements from the *NSReq* to the service access points in the *NFO*. Therefore, in this first step, we transform each *SAPR* element into a *SAP* element in the *SM*. The *SAP Functional Characteristic* elements of a *SAP* in the *NFO* are the corresponding elements of the *Accessed Functionality* elements in the *NSReq*. These two elements provide the same information about the *SAP*s and *SAPR*s, respectively, as shown in the *NSReq* and *NFO* metamodels. Therefore, we transform each of the former elements to the latter in the *SM* model. Then, we associate each created *SAP Functional Characteristic* with its corresponding *SAP* in the *SM* according to the *SAPR*s. We use the *TempRef* associations for this association, as shown in the *SM* metamodel. It is a temporary association, and in Step 2, we replace it by *SAP Interface* elements after realizing the interfaces that each *SAP* in the *SM* should expose according to the *NFO*.

Non-functional Requirements: *NFR*s are out of the scope of the *NFO.* Therefore, we transform the *NFR*s of the *NSReq* into the same elements in the *SM*, i.e. *NFR* elements, so we can use them for dimensioning – discussed in Chapter 5. We associate each transformed *NFR* with its corresponding *SAP Functional Characteristic* element in the *SM*.

Tagging the elements: In different steps of the approach, we will add different elements to the *SM* from different sources. The original source of the *SM* elements and their matching status affect the way we process them. Therefore, we need such information on each *SM* element. At different steps, we tag these elements according to their condition using their '*matchingTag*' attribute – this attribute is an enumeration type ('*MatchingTag*'). These elements include functionalities, their decomposition and dependencies, architectural blocks and their dependencies, and SAP functional characteristics, as shown in the *SM* metamodel. We use three tags including '*matched*', '*unmatched*', and '*from ontology*'. If we add a new element from the *NSReq* to the *SM* we tag it as '*unmatched'*. When we match an *SM*'s '*unmatched*' element to an *NFO*'s element we change its tag to '*matched'*. If we add an element from the *NFO* to the *SM*, we tag it as '*from ontology'*. At Step 1 (Initialization of the SM model), we tag all the elements that we create in the *SM* as '*unmatched'*. An '*unmatched*' element at this step means it is originally from the *NSReq*, and it is not mapped to the *NFO* elements. The *NFR* elements are exempt from tagging, as their only source is the *NSReq*, and we do not map them to elements of other models.

4.2.2 Step 2 – Decomposing the SM model

According to Section 4.2.1, at this point, the *SM* contains only the requirements provided in the *NSReq*. In this step, we map the *SM* elements to their corresponding elements in the *NFO* to find their matching elements. Then, we decompose/enrich each matched element in the *SM* according to its matched element in the *NFO*. This helps us to fill the gap between the requirements and the desired *NSD*, as discussed earlier.

The functional requirements in the *NSReq* are the main requirements that describe the required network service. In order to map the *SM* elements, we traverse the functional hierarchy in the *SM*. From each traversed functionality, we traverse its related architectural blocks. We map each traversed element with an '*unmatched*' tag to their corresponding *NFO* elements.

Traversing the *SM* hierarchies is a one-time procedure, and we call it *SM Traversal*. Every time we match an element in the *SM Traversal* procedure, we capture its related elements into the *SM* using specific procedures. These procedures are called *Functional Capturing* and *Architectural Capturing* respectively for functionalities and architectural blocks. In these procedures, we may add new functionalities and/or architectural blocks into the *SM*. Every time we add a functionality or architectural block, we run the appropriate capturing procedure for it. Therefore, these capturing procedures are run recursively, and they capture all the *NFO* sub-trees related to the *NSReq* into the *SM*. Figure 4.3 shows an overview of Step 2.

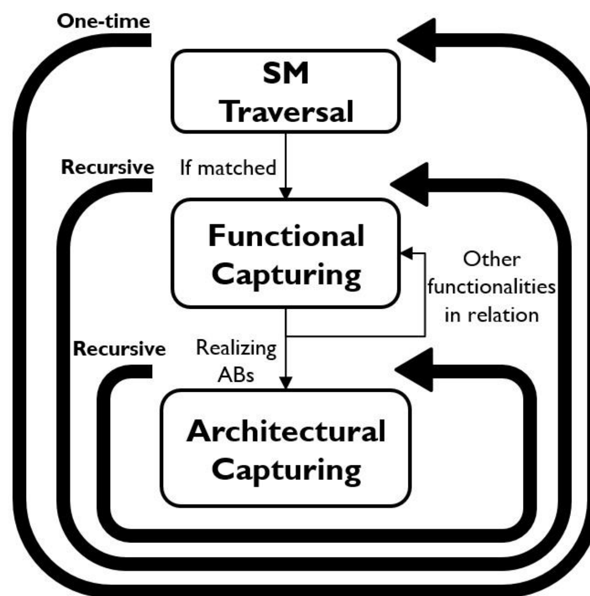

Figure 4.3 – Overall view of decomposing the SM model

The required *SAP*s and their characteristics are defined in the *NSReq*, and we have captured them in the *SM*. In standard architectures, not all of the interfaces matching the required *SAP*s characteristics are exposed to the environment. Therefore, we map all the *SAP*s in the *SM* to the *NFO SAPs* in order to find the interfaces they should expose. We do so in a one-time procedure referred to as *SAP Capturing*. This procedure is done after traversing the *SM* hierarchies completely.

4.2.2.1 *SM Traversal*

SM Traversal procedure is composed of two sub-procedures, main sub-procedure and special sub-procedure. The main sub-procedure is run in all *NSReq* cases. The special sub-procedure is run for the *NSReq* special cases, i.e. *NSReq* without *FR*. According to Section 3.1.2, in such cases, all the functionalities that the required architectural blocks realize in the *NFO* are required. Therefore, the special sub-procedure takes into account these functionalities.

Main sub-procedure: In this procedure, we traverse the whole hierarchy of functionalities in the *SM* using a breadth-first traversal. For each traversed functionality with the '*unmatched*' tag, we traverse the functionality hierarchy of the *NFO* to find the matching functionality. As discussed earlier, we specify a functionality with its name and aliases. Therefore, two functionalities match if they have the same name or at least one of their aliases matches the name or an alias of the other. If there is a match, we tag the *SM*'s functionality as '*matched*', and we run the *Functional Capturing* procedure for it. All the *SM*'s functionalities that were not matched in this procedure remain with the '*unmatched*' tag. This means we do not have information on these functionalities in the *NFO*.

Special sub-procedure: For an *NSReq* with the special case, we need to add all the functionalities realized by the required architectural blocks to the *SM*. For such an *NSReq*, at the beginning of the *SM Traversal* procedure, we traverse the *SM*'s architectural hierarchy. For each architectural block, we traverse the *NFO*'s architectural hierarchy to find the matching architectural block. Architectural blocks match to each other in the same way as the functionalities match, i.e. by names or aliases. For each matched architectural block, we add all the functionalities that it realizes in the *NFO* into the *SM*. We tag each added functionality as '*unmatched*' and associate it with the architectural block using a '*Realized By*' association in the *SM*. After

traversing the *SM*'s architectural hierarchy, all the required functionalities exist in the *SM*. So, we run the main sub-procedure of *SM Traversal*, i.e. traversing the *SM*'s functional hierarchy.

4.2.2.2 *Functional Capturing*

In this procedure, we capture all the relations and elements related to **a matched functionality** in the *NFO* that are relevant to our requirements. Then, we add them to the *SM* if they do not already exist there. These relations include the relations to parents, dependency, decomposition, specialization, and realization relations.

We start with capturing the relation to the parents, as we need these relations for checking the context of the dependency relations. In capturing the realization relations, we add the architectural blocks. Capturing the realization relations before the decomposition relations build the architectural hierarchy top-down. This simplifies the *Architectural Capturing* procedure compared to building this hierarchy bottom-up. *Functional Capturing* procedure has four main steps:

- Step 1 – Setting the Parents
- Step 2 – Capturing the Realization Relations
- Step 3 – Capturing the Dependency Relations
- Step 4 – Capturing the Decomposition/Specialization Relations

Step 1 – Setting the Parents: As discussed earlier, in SM Traversal procedure we traverse the functionalities top-down. Therefore, all the parents of **the functionality** that are relevant to the functional requirements have been added to the *SM* before this step. In this step, we only set **the functionality**'s relations to its parents in the *SM*, and we do not add other parents.

At first, we check whether the parent(s) of **the functionality** in the *NFO* exits in the *SM*. For each of the existing parents in the *SM*, we create a '*Composed Of*' relation between **the**

functionality and the parent, if such relation does not already exist. Then we tag this relation as '*from ontology*'. If such relation already exists with *'unmatched'* tag we change it to '*matched'*.

Step 2 – Capturing the Realization Relations: In this step, we capture the architectural blocks that realize **the functionality** according to its architectural constraints. First, we capture all the architectural blocks through the '*Realized By*' associations of **the functionality** in the *NFO*. Then we check them against **the functionality**'s architectural constraints. As discussed in Section 4.2.1, we had transformed the *AR*s in the *NSReq* into *AB* elements in the *SM* with the '*unmatched*' tag. Therefore, these *AB*s are the architectural constraints for their associated functionalities in the *SM*.

If each captured architectural block already exists in the *SM* with an '*unmatched*' tag and associated with **the functionality**, it means it complies with the constraints. We run the *Architectural Capturing* procedure for it, and we change its tag to '*matched*'. If a captured architectural block does not exist in the *SM*, we check it against **the functionality**'s parents'/ancestors' constraints. According to Section 3.2.2, if the architectural block in the *NFO* is the child of the architectural blocks specified by these constraints, it complies with the constraints. We add it to the *SM*, run the *Architectural Capturing* procedure for it and tag it as '*from ontology*'. Then, we create a '*Realized By*' association between **the functionality** and this architectural block.

Step 3 – Capturing the Dependency Relations: In this step, we capture **the functionality**'s dependency relations in the *NFO* with a valid context. For each dependency, first, we check whether it already exists in the *SM*. If it exists and it is tagged as '*unmatched'*, we change its tag to '*matched'*.

If the dependency does not exist in the *SM*, we check whether its context is valid in the *SM* – refer to Sections 3.2.1 and 3.2.4. If the context is valid we should add the dependency to the

SM. The pre-condition for this is that the dependency's supplier functionality should exist in the *SM*, and it should have been mapped to *NFO* functionalities. If it exists with an *'unmatched'* tag, it means it has not been mapped to the *NFO* elements. We run the *Functional Capturing* procedure for it, and change its tag to *'matched'*. If it does not exist in the *SM*, we add it and run the *Functional Capturing* procedure for it. Then we tag it as *'from* ontology'. After the pre-condition is met, we add the dependency to the *SM* and tag it as '*from ontology*'.

Step 4 - Capturing the Decomposition/Specialization Relations: In this step, we capture the decomposition and specialization relations of **the functionality** form the *NFO*. For the decomposition relations, we only take the mandatory ones into consideration. The optional decompositions should have been required in the *NSReq*, and we take them into account in the *SM Traversal* procedure. We check whether each mandatory decomposition and specialization relation of **the functionality** in the *NFO* exists in the *SM* with the '*unmatched*' tag. If yes, we change its tag to '*matched*'.

If not, first, we check whether the child functionality in this relationship exists in the *SM*. If it exists with the '*unmatched*' tag, we run the *Functional Capturing* procedure for it, and we change its tag to '*matched*'. If the child functionality does not exist in the *SM* we add it; we run the *Functional Capturing* procedure for it, and we tag it as '*from ontology*'. Then, we add the decomposition relation to the *SM*, and we tag it as '*from ontology*'.

4.2.2.3 *Architectural Capturing*

In this procedure, we capture the relations of **an architectural block** in the *NFO* that are relevant to our requirements. Then, we add them to the *SM* if they do not already exist there. These relations include the relations to the parents and the dependency relations.

We add the architectural blocks into the *SM* through the functionalities' realization relations in the *Functional Capturing* procedure. Also, in this procedure, we set the architectural blocks' relations with their parents. Therefore, the decomposition of each architectural block is captured into the *SM* indirectly, and we do not need to capture the decomposition relations. We start this procedure by setting the relationship with parents, as we need to check these relations for validating the dependencies' contexts. *Architectural Capturing* procedure has three steps:

- Step 1 – Setting the Parents
- Step 2 – Capturing the Interfaces
- Step 3 – Capturing the Dependency Relations

Step 1 – Setting the Parents: In this step, we set **the architectural block**'s relations with its parents in the *SM* according to the *NFO*. An architectural block is in relation with its parents through *Architectural Composition* elements, as shown in the *NFO* and *SM* metamodels. For each parent of **the architectural block** in the *NFO* that also exists in the *SM*, we capture its *Architectural Composition* elements that are associated with **the architectural block**. For each captured *Architectural Composition* that also exists in the *SM*, we associate the architectural block with it in the *SM*, if it is not already associated. If the captured *Architectural Composition* does not exist in the *SM*, we add it to the *SM*, and we associate **the architectural block** and the parent to it.

Step 2 – Capturing Interfaces: In this step, we capture **the architectural block**'s interfaces. We add all the architectural block's *Interface* elements from the *NFO* into the *SM*.

Step 3 – Capturing the Dependency Relations: In this step, we capture the dependency relations related to **the architectural block** in the *NFO* with a valid context. For each dependency that **the architectural block** is associated with, whether it is the dependency's client or supplier, we need to capture the dependency. For such a dependency, we check the dependency's contexts

validity – refer to Sections 3.2.2 and 3.2.4. For each dependency with a valid context, we check whether it exists in the *SM*. If it exists with an '*unmatched*' tag we change its tag to *'matched'*.

If it does not, the pre-condition for adding it into the *SM* is that both of its client and supplier *AB*s should exist in the *SM*. If the architectural block on the other side of the dependency exists in the *SM* with an '*unmatched*' tag, we run the *Architectural Capturing* procedure for it and change its tag to *'matched'*. If the pre-condition is met, we add the dependency and tag it as *'from ontology'*. In either case (the dependency already existed or we just added it), we need to enrich the dependency by its characteristics specified in the *NFO*. Therefore, we add the dependency's *ADep Interfaces* elements from the *NFO* into the *SM* in association with the dependency.

4.2.2.4 *SAP Capturing*

As discussed earlier, one of the goals of this procedure is to add the information about the interfaces exposed by the *SAP*s from the *NFO* to the *SM*. First, we map the *SM SAP*s to the *NFO SAPs* to find their match and set their exposed interfaces accordingly. The other goal is to project the *SAP*s in the *SM* to all the functional levels so we can use them, later in Section 4.2.5, for all the designed *NSD*s.

Setting the SAPs exposed interfaces: In this part of the *SAP Capturing* procedure, we map each *SAP* in the *SM* to a group of the *SAP*s in the *NFO*. The *SAP*s in this group should have a valid context in the *SM* – see Sections 3.2.3 and 3.2.4 – and should be associated with the architectural blocks that also exist in the *SM*. We only consider these *SAP*s for mapping as the rest of the *SAP*s in the *NFO* are irrelevant to our requirements. We match the *SAP*s according to their exposed functionalities, i.e. their *SAP Functional Characteristic* elements. We map each of the *SAP Functional Characteristic* elements of the *SM*'s *SAP*s to the same elements of the captured *NFO SAP*s in the group. These two elements match if their functionalities and planes

are the same, and the roles of the one from the *NFO* includes all the roles of the one from the *SM*. If there are any matches, we change the tag of the *SAP Functional Characteristic* element in the *SM* to *'matched'*.

For each match, we add the associated *SAP Interface* element from the *NFO* into the *SM*. We associate it with the matching *SAP* and *SAP Functional Characteristic* in the *SM*. Then, we associate the *SAP* with the architectural block that has the interface referenced by the *SAP Interface* element. Each *SAP* in the *SM* should have a matching interface for each of its *SAP Functional Characteristic* elements. If all the *SAP Functional Characteristic* elements of a *SAP* in the *SM* are tagged as '*matched*' we tag the *SAP* as '*matched*' as well.

For the *SAP Functional Characteristic* elements in the *SM* that have remained '*unmatched*', there is no information in the *NFO*. For each of them, we set all the interfaces in the *SM* that match its characteristics as its exposed interfaces. An interface matches a *SAP Functional Characteristic* element if both have the same plane and functionality, and the interface's roles include all the *SAP Functional Characteristic*'s roles. For each matched interface, we create a *SAP Interface* element that references the interface. We associate it with the matched *SAP Functional Characteristic* and its related *SAP*. Then we change the tag of the matched *SAP Functional Characteristic*, and if applicable the tag of the *SAP*, to '*matched*'.

SAP Projection: The VNFs that we use to generate generic *NSD*s may realize functionalities at any level in the functional hierarchy. Therefore, the functionalities that a VNF exposes through its interfaces also can be at any level. However, at this point, the *SAP*s in the *SM* expose functionalities at specific levels that are not necessarily the same level as the VNFs' functionalities levels. As a result, we should design the *SAP*s in a way that we can use them for the VNFs with functionalities at any level. To achieve that, we project all the *SAP*s to all the functional levels in the *SM*.

In *SAP Projection*, we consider all the *SAP*s in the *SM* with the '*matched*' tag. For each functionality that these *SAP*s expose, all of its parent and ancestors that are single-flow should be exposed by that *SAP* as well – single-flow functionalities are discussed in Section 4.2.5.5. This projects the *SAP* to the higher levels of functionality in the *SM*. We should project the *SAP* to the lower levels of functionality as well. The child of a functionality that has no incoming dependency is at the beginning of the dependency chain. Therefore, such functionality sends requests to other functionalities in the decomposition. Therefore, if the environment needs to communicate with the functionalities at this functional level, it should communicate with this functionality. Thus, the *SAP*s should expose the children and the grandchildren with no incoming dependency of their exposed functionalities as well. The grandchildren of a functionality are the children of the functionality's children which recursively goes to the leaves.

Accordingly, we create a *SAP Functional Characteristic* element for each of the aforementioned parents, ancestors, and children functionalities. These are the projected *SAP Functional Characteristic* elements. The plane and the roles of these projected elements are the same as the plane and the roles of their originating *SAP Functional Characteristic*s. We associate the projected *SAP Functional Characteristic*s with their corresponding *SAP*s in the *SM* using the *TempRef* association, as shown in the *SM* metamodel.

4.2.3 Step 3 – Selecting the VNFs

In this step, we select the set of VNFs from the *VNF Catalog* which contains the *VNF Package Info* elements. As discussed earlier, the VNFs are the implementations of the architectural blocks. Therefore, we select the VNFs that match the architectural blocks in the *SM*. The '*Implemented Arch. Block*' attribute of the *VNFAD* element of a VNF specifies the architectural blocks it realizes, as discussed in Section 3.3.2. Therefore, an *AB* element in the *SM* matches a VNF when its name or one of its aliases matches this attribute of the VNF.

We check all the VNFs in the *VNF Catalog* against each *AB* element in the *SM*. The VNFs that match one or more *AB*s in the *SM* belong to the set of selected VNFs for *NSD* design. We add the *VNF Package Info* element of each of these VNFs into the *SM* if it is not already there. We associate each added *VNF Package Info* element with all of its matching *AB*s in the *SM*.

4.2.4 Step 4 – Generation of forwarding graphs

In this step, we generate all the forwarding graphs that realize the required network service. To do so, first, we capture all the combinations of the functionalities that realize the functional aspects of the network service. Accordingly, we capture all the combinations of the architectural blocks that realize the network service. Then, we generate all the forwarding graphs composed of VNFs accordingly.

A combination of the *SM* functionalities that all together compose the root functionality of the *SM* represents the functional aspect of the required network service. We call such a combination an *FFG*. The root functionality in the *SM* hierarchy is also an *FFG*. A parent and its child cannot exist in the same *FFG,* as it is redundant. In an *FFG*, we can substitute a functionality with all of its children or grandchildren at any level, and it results in another *FFG*. We consider this combinatorial aspect to generate all the *FFG*s from the *SM* hierarchy.

A combination of the architectural blocks that realize all the functionalities in an *FFG* realize the required network service. We call this combination an *AFG*. In an *AFG*, the sequence of the architectural blocks is according to their dependencies in the *SM*. This sequence defines the VNFs' connectivity in the forwarding graph later on. We generate all the possible *AFG*s from each of the generated *FFG*s. Multiple architectural blocks may realize a functionality in the *FFG*, and it results in multiple *AFG*s. We consider this combinatorial aspect to generate all the possible *AFG*s.

We generate a forwarding graph by substituting an *AFG*'s architectural blocks with the VNFs realizing them. Such forwarding graphs specify the VNFs' connectivity according to the *AFG*'s sequence. Therefore, we can generate an *NSD* and its *VNFFGD*s from each forwarding graph. We call such forwarding graph a *Pre-VNFFG,* as it does not specify the virtual links yet. Multiple VNFs may realize an architectural block in the *AFG*, and it results in multiple *Pre-VNFFG*s. By considering this combinatorial aspect we generate all the possible *Pre-VNFFG*s from each *AFG*.

As an example, in an *SM* model, the root functionality is decomposed to two other functionalities. Three different architectural blocks realize each of these functionalities, and two different VNFs realize each architectural block. Therefore, we can generate two *FFG*s for the required network service – one *FFG* is composed of the root, and the other is composed of the root's decomposition. We can generate three *AFG*s from the first *FFG* and nine *AFG*s from the second *FFG*. We can generate two *Pre-VNFFG*s from each *AFG* of the first *FFG* and four *Pre-VNFFG*s from each *AFG* of the second *FFG*. In total, we can generate 42 *Pre-VNFFG*s.

FFG generation: This is a recursive process. We invoke each recursive step for a functionality in the *SM* hierarchy, and we call this functionality the *sub-root*. The goal of each recursive step is to find the *FFG*s in the sub-tree of the *sub-root*. We call these *FFG*s *partial FFG*. The combination of functionalities in each *partial FFG* composes the *sub-root* functionality. The initial recursive step starts from the *SM* root functionality. As discussed earlier, the combination of the functionalities in an *FFG* composes the root functionality. Therefore, the result of the whole recursive process is the set of all the *FFG*s in the *SM*. Each recursive step, at first, invokes a recursive step for each of the *sub-root*'s children. In the end, it returns all the *partial FFGs* of the *sub-root*.

The only *partial FFG* of a leaf functionality is itself. Therefore, when the *sub-root* is a leaf functionality, the recursive step returns only one *partial FFG* which is the *sub-root*. When the *sub-root* is a non-leaf functionality, each *partial FFG* is the concatenation of a *partial FFG* of each of the *sub-root'*s children. Therefore, all these combinations in addition to the *sub-root* itself (as a *partial FFG*) are the result of each recursive step.

In the *SM*, there might be some functionalities for which there is no VNF to realize them. We exclude the *FFG*s that contain such functionalities as we cannot generate any *Pre-VNFFG* from them later on. For each of the remaining *FFG*s, we generate an *FFG* element in the *SM* that references the *Functionality* elements accordingly.

AFG generation: In this process, we generate an *AFG* by substituting each functionality in an *FFG* with one or a group of architectural blocks realizing that functionality in the *SM*. This (these) architectural block(s) should directly realize the functionality, i.e. being in a '*Realized By*' association with the functionality. A group of architectural blocks realizes a functionality directly when they are in a chain of dependencies. All the dependencies in this chain have the same context. This context is the '*Realized By*' association between the functionality and the architectural block at the beginning of this chain.

We run this process for each generated *FFG*, and we generate all possible *AFG*s by considering the combinatorial aspect. For each *AFG* we generate an *AFG* element in the *SM* for it. Then, we reference all the architectural blocks in the *AFG* element accordingly. We preserve the information of the mapping between the architectural blocks and the *FFG*'s functionalities in the *AFG*, as we need it later on. We reference all the dependencies between the architectural blocks in the *AFG* with a valid context in the *AFG*. In an *AFG*, a context is valid only if its specified '*Realized By*' association exists in the *AFG*. It means the architectural block and the functionality in this association have a mapping in the *AFG*.

Earlier, we have projected the *SAP*s to all functional levels in the *SM*. In each *AFG*, architectural blocks expose functionalities at specific levels, and *SAP*s expose them by their *SAP Functional Characteristic* elements. Therefore, in the *AFG*, we reference the *SAP Functional Characteristic* elements that are related to the *AFG*'s architectural blocks by their interfaces.

In *AFG* generation form an *FFG*, we consider all the combinations of architectural blocks that realize the *FFG*'s functionalities. In some of these combinations, there are architectural blocks from different architectural compositions. Therefore, there is no information regarding the dependencies between these architectural blocks. We refer to such *AFG*s as incomplete. We infer the missing dependencies in such *AFG*s in a process called *AFG Completion*. In this process, we may complete some of these *AFG*s, and we exclude the remaining incomplete ones.

***AFG Completion**:* As discussed earlier, each incomplete *AFG* is composed of portions of different architectural compositions. Each of these architectural compositions has some architectural blocks that are not involved in the *AFG*. The dependencies between these architectural blocks and the ones involved in the *AFG* are missing in the *AFG*.

If two dependencies have the same client and supplier functionalities in their *ADep Interface* elements they match the same functional dependency. We call such dependencies equivalent, and their client and supplier architectural blocks equivalent as well. Two equivalent dependencies are not in the same architectural composition.

In an *AFG*, each missing dependency has an equivalent dependency which is also missing in that *AFG*. We need to create one dependency for each pair of equivalent missing dependencies in an *AFG*. In order to create such a dependency, the *ADep Interfaces* elements of both missing dependencies should match. They match if they have the same plane, client and supplier functionalities, roles and protocols. If so, we create a dependency between the

two missing dependencies' client and supplier architectural blocks that are involved in the *AFG*. As discussed earlier, only one of the two clients and one of the two suppliers are involved in the *AFG* for each missing dependency. The created dependency is only for this *AFG*, therefore, we keep a reference from the dependency to the *AFG*. If the *ADep Interfaces* elements of a pair of missing dependencies do not match, we do not create any dependency. Therefore, the *AFG* remains incomplete.

Pre-VNFFG generation: *Pre-VNFFG* generation is similar to *AFG* generation. We generate a *Pre-VNFFG* by substituting each architectural block in an *AFG* with a VNF realizing the architectural block in the *SM*. We run this process for each of the generated *AFG*s. A combination of VNFs realizing each *AFG*'s architectural block forms a *Pre-VNFFG*. We create a *Pre-VNFFG* element in the *SM* for each of these combinations. We keep a reference in the *Pre-VNFFG* element to the *VNF Package Info* elements of the VNFs in the combination. We also keep the information on the mapping between the VNFs and the architectural blocks in the *Pre-VNFFG* element, as we need it in the next step. To access the related dependencies and *SAP*s we use the references in the *AFG*, and we do not reference them in the *Pre-VNFFG*.

4.2.5 Step 5 – Generation of Network Service Descriptors

In this step, we generate a generic *NSD* model from each of the *Pre-VNFFG*s in the *SM*. In each *NSD* model, we add the *VNFD*s, and we create the *VLD*, *SAPD*, *VNFFGD*, and *NS Deployment Flavor* elements. In the *SM*, we use an *SmNsd* element to keep a reference between a *Pre-VNFFG* and its *NSD*.

4.2.5.1 *Adding the VNFDs*

We reference each *VNFD* of the *Pre-VNFFG* in the *NSD* – *VNFD*s are entailed by the *VNF Package Info* elements in the *Pre-VNFFG*. Then, we create a *VNF Profile* element for each referenced *VNFD* - We will use *VNF Profile*s for the VNFs' connectivity and dimensioning.

4.2.5.2 *Creating the VLDs*

Virtual links in the network service connect VNFs to other VNFs or to SAPs. We design the VLs of the former and the latter case according to the architectural dependencies and the SAP elements in the *AFG* respectively.

According to Section 3.2.2, an architectural dependency may have multiple *ADep Interfaces* elements. By these elements, the dependency specifies multiple interfaces of its client and supplier architectural blocks for their communication. We call the VNFs corresponding to the client and supplier ABs the client and supplier VNFs of the dependency respectively. Therefore, the architectural dependency specifies the interfaces of these VNFs as well. Similarly, a *SAP* in the *SM* specifies one or multiple interfaces of its related architectural blocks and their corresponding VNFs too. Each VNF interface is exposed by a VnfExtCp. In order to design the VLs and generate their *VLD*s, we should figure out the VnfExtCps corresponding to the architectural dependencies and the *SAP*s.

Creating the VLDs for VNFs communication: As discussed earlier, we find the VnfExtCps corresponding to each architectural dependency in the *Pre-VNFFG*. To do so, we find the interfaces of the client and supplier VNFs that match the specified interfaces for the architectural blocks. These interfaces match if they have the same name and matching *Functional Characteristic* elements (all their attributes should match). The VnfExtCps associated with the matched VNF interfaces – according to the *VNFAD* – are the VnfExtCps related to the dependency.

For the dependencies with related VnfExtCps in common, there should be an *E-LAN* virtual link in the network service to connect their VnfExtCps. For the rest of the dependencies which have two related VnfExtCps, an *E-Line* virtual link should connect them. For creating the *VLD*s, first, we group the dependencies according to their related VnfExtCps in common. Consider each dependency as a node in a graph. Consider each related VnfExtCp that is common between two dependencies as an edge between the two nodes (those dependencies). Each connected graph [49] inside this graph defines a group of dependencies that have related VnfExtCps in common. No two dependencies in different groups have related VnfExtCps in common. We connect the VnfExtCps in each group with one virtual link.

We should check the compatibility of the protocols of the interfaces that communicate with each other according to the dependencies. Therefore, we check the compatibility of the protocols specified by each *ADep Interfaces* element for the source and the target interfaces. If the protocol of the source and the target interfaces are the same, they are compatible. Also, if two protocols are in the *IsServedBy* association directly or through other protocols in the *Protocol Stack* model, they are compatible. For each group, if all the interfaces in communication have compatible protocols, we create a *VLD* element. If not, we do not create the *VLD*, as some interfaces in the group cannot communicate. Therefore, we dismiss this *NSD*. If the number of VnfExtCps in the group is more than two we set the virtual link's flow pattern to *E-LAN*, otherwise we set the flow pattern to *E-Line*. We specify the flow pattern of a virtual link using the *VLD*'s '*ConnectivityType*' attribute as shown in the VNFD metamodel.

The layer protocols of a virtual link should be the same as the layer protocols of all the VnfExtCps it connects to. Therefore, we set the layer protocols of each *VLD* according to its VnfExtCps' layer protocols. For each *VLD* we create an *NsVlDf* and an *NsVl Profile* element, and we keep a reference to the *NsVlDf* in the *VLD* and the *NsVl Profile*. We need these two

elements for the purpose of the VNFs and virtual links connectivity. To set this connectivity, for each *VNFD* that has at least a *VnfExtCpd* associated with a *VLD* we create an *NsVirtual-LinkConnectivity* element. We reference the *VnfExtCpds* and the *NfVl Profile* of the *VLD* in this element. We reference this element in the *VNF Profile* of the *VNFD*.

We create the *VLD*s for the second case (connection between the SAPs and the VNFs) while creating the *SAPD* elements. We will discuss it in the following section.

4.2.5.3 *Creating the SAPDs*

In this sub-step, we create a *SAPD* element for each *SAP* in the *SM*. For each *SAP*, first, we realize its related VnfExtCps according to the interfaces that the *SAP* specifies. A *SAP* specifies the interfaces by its *SAP Functional Characteristic* elements. Some of these elements of each *SAP* are related to the *AFG*, and therefore to the *NSD*. For each *SAP*, we find the interfaces of the VNFs in the *Pre-VNFFG* that match the interfaces specified by the *SAP Functional Characteristic* elements in the *AFG*. The *SAP* should connect to the VnfExtCps associated with these VNF interfaces. Mapping a VNF interface to an architectural block interface was discussed in Section 4.2.5.2.

For each *SAP* in the *SM,* we create a *SAPD* element in the *NSD*. If there is one VnfExtCp that the *SAP* relates to, we reference its *VnfExtCpd* directly by the *SAPD*. If there are multiple VnfExtCps, the *SAP* should connect to them by a virtual link. Therefore, we create a *VLD* as discussed below. The layer protocols of the *SAPD* should be the same as the layer protocols of all the VnfExtCps it connects to. Therefore, we set each *SAPD*s' layer protocols accordingly.

Creating the VLDs for VNFs and SAPs communication: First, we create a *VLD* element. If there is one VnfExtCp related to the *SAP* we set the *VLD*'s flow pattern as *E-Line* since the other connection of the virtual link is the *SAP*. If there are more than one related VnfExtCps we

set the flow pattern to *E-LAN*. Same as the first case of *VLD* creation, we create an *NsVlDf*, an *NsVlProfile*, and a *NsVirtualLinkConnectivity* element for the *VLD*. We add their references in the same way as discussed in the first case as well. In addition, we reference the *SAPD* in the *NsVirtualLinkConnectivity* element too.

4.2.5.4 ***Creating the VNFFGDs***

In this step, we generate the *VNFFGD*s for the *NSD* according to the *Pre-VNFFG*. We create one *VNFFGD* for each plane, as we want to keep the planes separate from each other. Each *VNFFGD* references the *VNFD*s and their *VnfExtCpd*s, *VLD*s, and *SAPD*s which are involved in the plane of the *VNFFGD*. To create the *VNFFGD*s we group these elements based on their planes, and we create one *VNFFGD* element according to each group. These elements can be redundant in different groups, as they may involve in different planes.

We distinguish the *VLD*s' and *SAPD*s' planes according to the VNF interfaces' planes they are related to, as discussed in Sections 4.2.5.2 and 4.2.5.3. We distinguish the *VnfExtCpd*s' planes based on the VNF interfaces they expose in the network service. A *VnfExtCpd* may expose other VNF interfaces that are not involved in the network service, and they are not considered. A VNF interface is involved in the network service if it is involved in a communication, i.e. related to a *VLD* or a *SAPD*. We distinguish the *VNFD*s' planes also according to their interfaces involved in the network service.

From each created *VNFFGD* element, we reference the *VNFD*s and *VLD*s in each group. Then, for each *VNFD* in the group, we create a *CpdPool* element for the *VNFFGD* that references all the *VNFD*'s *VnfExtCpd*s in the group. We also create a *CpdPool* that references all the *SAPD*s in the group. We create the *NFPD*s for the *VNFFGD* based on the flows in the network service, as discussed in the following section.

4.2.5.5 *Creating the NFPDs*

As discussed in Section 2.2.1, each NFP is a sequence of VnfExtCps in a VNFFG that defines a path for specific packet flows. We define different paths based on the flows for different functionalities in the network service. We call these flows the *Propagation Flow*s, and we design the NFPs based on them.

Propagation Flow: A *Propagation Flow* is a sequence of VNF interfaces through which the packet flow related to a functionality on a specific plane propagates in the network service, and interacts with the environment. We define the *Propagation Flow* based on this functionality on the specific plane. A *Propagation Flow* may involve other functionalities depending on their dependencies to this functionality. A *flow* starts from a VNF interface that exposes the *flow*'s functionality on the *flow*'s plane. Since the *flow* interacts with the environment, a SAP should expose this interface to the environment. We call such interface the *flow*'s starting interface. We determine the rest of the interface sequence of the *flow* according to the VNFs' dependencies and the flows inside the VNFs. We may use the term '*flow*' instead of '*Propagation Flow*' in this document for simplicity.

Single-flow functionality: A single-flow functionality is a functionality which in its decomposition, there is only one functionality that defines a *flow* per plane. As a result, a single-flow functionality is only composed of other single-flow functionalities. The functionalities that VNF interfaces expose are only single-flow. Therefore, the functionalities that SAPs expose are only single-flow as well. As a result, we define a *flow* only based on a single-flow functionality. Also, we conclude that the functionalities that a *SAPR* exposes in the *NSReq* are only single-flow.

In the *SM* example depicted in Figure 4.4, the *F2* and *F3* are single-flow since *SAPR1* and *SAPR2* exposes them respectively. Therefore, their children are also single-flow, i.e. *F5*, *F6*,

F7, and *F8*. On the other hand, *F1* is not single-flow since it has two functionalities that define *flow*s in its decomposition, as they are exposed by *SAP*s.

Propagation Flow design: To design the *flow*s in the network service, first we determine the functionalities on different planes in the *FFG* that are exposed by a SAP. Then, we determine the starting interface and the interface sequence of each potential *flow* accordingly. An interface may appear in different points of a *flow*'s sequence. Each time an interface appears in the *flow* a subset of its characteristics (functionalities, planes, and roles) is related to the *flow*. By mapping these characteristics to the *ADep Interfaces* elements in the *AFG* and VNF's *Flow Transformation* elements of the *VNFAD*s in the *Pre-VNFFG* we can determine the *flow*'s sequence.

Determining the functionality and the plane of the flow: As discussed earlier, each functionality on a specific plane that a *SAP* in the *AFG* exposes defines a *flow* in the network service. The information on which functionalities a *SAP* exposes exists in the *SAP Functional Characteristic* elements as shown in *NFO* and *SM* metamodels. Therefore, we create a *Propagation Flow* element in the *SM* for each of these functionalities on the specified planes. We reference the functionality, the plane, and the *SAP* in the *Propagation Flow* element.

In the next sub-steps, each time we determine the appearance of an interface in the *flow*'s sequence we create an *SmInterface* element in the *SM*. We associate it with the *Propagation Flow* element. We set its attributes according to the subset of the interface's characteristics related to the *flow* (functionality, plane, and roles). Finally, we reference the source that we determined this appearance based on, i.e. the *Flow Transformation* or the *ADep Interfaces* element. For details of the *Propagation Flow* and the *SmInterface* elements in the *SM* refer to Figure 3.10.

Finding the starting interface of the flow: The VNF interface that is exposed by the *SAP* related to the *flow* and has the same functionality and plane as the *flow* is the *flow*'s starting

interface. The direction of the flows in a VNF defined by the *Flow Transformation* elements is always from the server/input interface to the client/output interface, i.e. the source interface is server/input, and the target interface is client/output. If the starting interface of a *flow* in the NS has the server and/or the input roles the *flow* is in the direction of the *Flow Transformation*s. Otherwise, the *flow* is in the opposite direction of the *Flow Transformations*. We call this backward propagation. We avoid designing *flow*s in backward propagation to avoid complexity. Therefore, if a starting interface has both server/input and client/output roles, we design the *flow* based on the server/input roles. If the starting interface has only the client and/or output roles, we inevitably design the *flow* in backward propagation.

In some cases, more than one *SAP* may expose a functionality on a specific plane but with different roles. In such a case, there is more than one candidate interface to select from as the *flow*'s starting interface. Among them, we select the interface with the server and/or input roles in order to avoid backward propagation.

Finding the interfaces sequence of the flow: We call the interface which a *flow* enters a VNF through an *entry interface*. We call the interface which a *flow* exits a VNF through an *exit interface*. Therefore, in a *flow*, there is always an *exit interface* after an *entry interface* and vice versa. The starting interface of the flow is always an *entry interface*.

Each time we determine the appearance of an interface in the *flow* we call it the *current interface*, including the starting interface. We find the next interfaces in the sequence according to the *current interface* by using two different procedures. If the *current interface* is an *entry interface* we use the '*Finding the next exit interface*' procedure to find the next interface(s). If it is an *exit interface*, we use the '*Finding the next entry interface*' procedure. Since the *flow*'s starting interface is always an *entry interface* we start with the former procedure.

Finding the next exit interface: The next *exit interface*(s) is (are) in the same VNF as the *current interface*. The *Flow Transformation* elements of a VNF determine the flows inside the VNF. Therefore, we find the next *exit interface*(s) using the current VNF's *Flow Transformation*s that are related to the *flow*. A *Flow Transformation* related to the *flow* should determine the *current interface* as one of its interfaces (source/target). In addition, the characteristics (functionality, plane, and roles) it defines for the *current interface* should match the subset of the *current interface*'s characteristics related to the flow. All the interfaces on the other side (target/source) of the related *Flow Transformation*s are the next *exit interface*s. For instance, if the *current interface* in a related *Flow Transformation* is the source interface, the interface on the other side is the target interface. Therefore, the target interface is the next *exit interface*.

The characteristics that each related *Flow Transformation* defines for the next *exit interface*s are their subset of characteristics related to the *flow*. We will use them to find the next *entry interface*(s) in the '*Finding the next entry interface*' procedure.

Special cases: In some cases, the *entry* and the *exit interface*s of a VNF in a *flow* are the same. If there are multiple dependencies associated with such an interface, there will be two cases for finding the next *entry interface*s. The first case is if the interface has the server and input roles on entry and has the output role on exit. This means that the VNF is responding to the incoming packet flow. Therefore, we need to design the *flow* in a way that it goes back towards the path it had come to the current VNF. It means we should use the same architectural dependency that we used for the incoming direction on the outgoing direction of the *flow* (in '*Finding the next entry interface*' procedure). The second case is if the interface has the server and client roles on the entry and exit respectively. This means the outgoing flow is a request to another interface with a server role. Therefore, in such a

case, we should avoid the architectural dependency used in the incoming direction for the outgoing direction.

Finding the next entry interface: The next *entry interface*(s) is (are) in the VNFs associated with the dependencies that are associated with the *current interface*. We find the next *entry interface*(s) according to the *ADep Interfaces* elements that are related to the *flow*. The related *ADep Interfaces* elements are the ones in which the characteristics (functionality, plane, and roles) defined for the *current interface* match the *current interface*'s characteristics related to the *flow*. If the current interface fits into the two special cases discussed previously, we should consider the guidelines mentioned for each case to find the related *ADep Interfaces*. The interfaces on the other side of the related *ADep Interfaces* elements are the next *entry interfaces*. The characteristics specified for each in these *ADep Interfaces* elements are their subset of characteristics related to the *flow*. We use these characteristics, except the roles, to find the next *exit interface*s.

An *ADep Interfaces* element may specify multiple roles for the interfaces on both sides (source and target). Therefore these interfaces may have input and output and/or client and server roles. As discussed before, the server/client pattern indicates the direction of the *flow*. Therefore, having multiple roles for both sides in an architectural dependency results in *flow*s in both directions. In order to find the next entry interface, and in general to define a flow, we should take only one direction. Among the next *entry interface*s' roles defined by the *ADep Interfaces* elements, we should select only the roles that are complementary of the *current interface*'s roles – as discussed before, server and input roles are complementary of client and output roles respectively, and vice versa.

Creating NFPDs from Propagation Flows: As discussed earlier, a *flow* is a sequence of VNF interfaces, and an NFP is a sequence of VnfExtCps. A VnfExtCp exposes one or many

VNF interfaces. Therefore, we derive the sequences of VnfExtCps, i.e. NFPs, from the sequences of interfaces (*flow*s) by substituting each interface in a *flow* with its associated VnfExtCp. It is possible that we derive identical NFPs from multiple *flow*s, as a VnfExtCp may expose multiple VNF interfaces. Therefore, among the derived NFPs, we exclude the redundant NFPs from the set of the results.

For each remaining NFP, we create an *NFPD* element in the *NSD*. We reference the related *VnfExtCpd*s by the *NFPD*. The VNFFG which has a *flow* also has the NFP derived from the *flow*. Therefore, we reference each *NFPD* by its associated *VNFFGD*.

4.2.5.6 *Creating the NS Deployment Flavor*

As discussed in Section 3.7.4, an *NsDf* element indicates the non-functional characteristics of the network service including QoS and scalability. It also specifies the composition and the connectivity inside the network service through the *VNF* and *NsVl Profile* elements. Therefore, we create an *NsDf* element for the *NSD*. We reference all the *VNF* and *NsVl Profile* elements that we have created in the previous sub-steps by this *NsDf*. We will enrich this *NsDf* by adding the QoS characteristics of the network service later on in Chapter 5.

4.2.6 An example of NSD generation

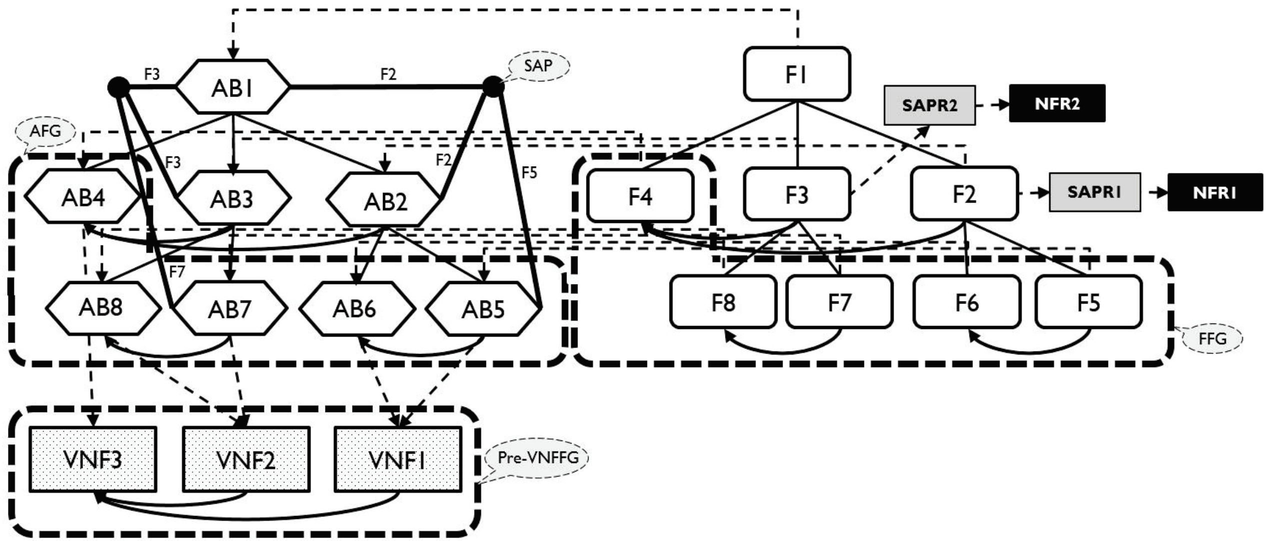

Figure 4.4 – An SM model example

Figure 4.4 shows an *SM* example, which is the result of the first four steps of the *NSD* generation process. This example is based on the *NSReq* and the *NFO* model examples in Figure 3.2 and Figure 3.4 respectively. In Step 2, *FR2* and *FR3* in the *NSReq* match with *F2* and *F3* in the *NFO*, respectively. Then, we decomposed *F2* and *F3* further according to the *NFO*. We have added the architectural blocks realizing the functionalities, and their *SAP*s according to the *SAPR*s. In the result of the *SAP Projection* process, we have associated the *SAP*s to other interfaces of the architectural blocks at other levels.

In Step 3, we have captured the VNFs which realize the architectural blocks in the *SM* model. In Step 4, we captured five potential *FFG*s including {*F1*}, {*F2*, *F3*, *F4*}, {*F5*, *F6*, *F3*, *F4*}, {*F2*, *F7*, *F8*, *F4*}, and {*F5*, *F6*, *F7*, *F8*, *F4*}. The captured VNFs only realize *F5*, *F6*, *F7*, *F8*, and *F4*. Therefore, the only remaining *FFG* is the last one. There is a one-to-one realization relation between the architectural blocks and the functionalities in the *SM*. Therefore, the potential *AFG*s are similar to the *FFG*s. The only remaining *AFG* is {*AB5*, *AB6*, *AB7*, *AB8*, *AB4*}. Therefore, the only *Pre-VNFFG* is {VNF1, VNF2, VNF3}.

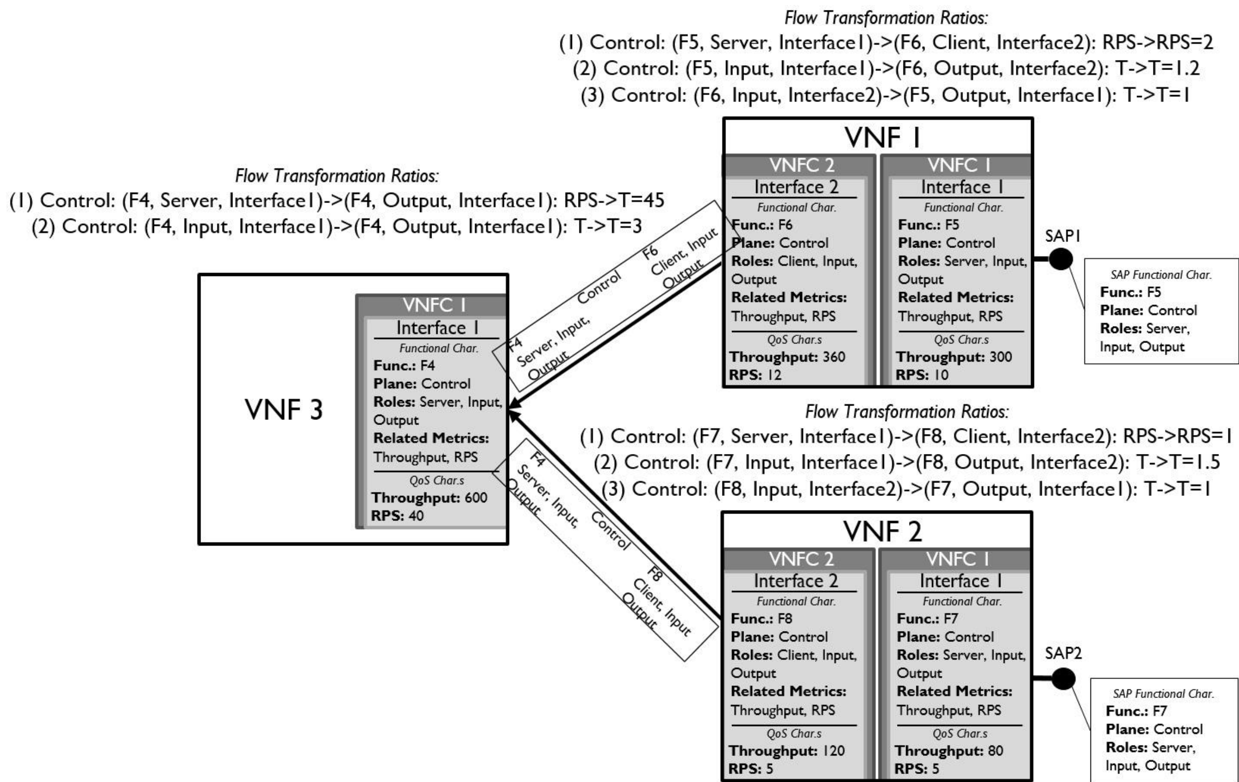

Figure 4.5 – Pre-VNFFG example with further details

Figure 4.5 shows a more detailed view of the *Pre-VNFFG* generated from the *SM* model in Figure 4.4. It shows the VNFs' dependencies with their *ADep Interfaces* elements and the *SAP*s with their *SAP Functional Characteristic* elements. For each VNF, its VNFCs and *Interface* elements are shown. Each VNF interface has a *Functional Characteristic* and two *QoS Characteristic* elements. For example, the *Interface 1* in the VNF1 provides access to the control plane of functionality *F5*, and it has server, input and output roles for that functionality. Its *QoS Characteristic*s are 300 units of throughput and 10 RPS.

In this *Pre-VNFFG*, the *ADep Interfaces* elements of each dependency are shown on top of it. For instance, the *ADep Interfaces* of the dependency at the top shows that the dependency is on the control plane. It shows VNF1 communicates through its *Interface 2* with the roles of

client, input, and output for functionality *F6*. VNF3 also communicates through its *Interface 1* with the roles of server, input, and output for functionality *F4*.

From Step 5 (*NSD* generation), we just discuss the *Propagation Flow* design example, since the rest of it is mostly about creating different elements and setting their attributes in the *NSD*.

Example of the Propagation Flow design: In the *SM* shown in Figure 4.4, there are two *SAP*s each of which exposes one functionality on the control plane. Therefore, we design two *Propagation Flow*s including *flow1* for *SAP1* and *flow2* for *SAP2*. The functionality of the *flow1* and *2* are *F5* and *F7* respectively. The plane of both *flow*s is control. *SAP1* exposes the VNF1-*Interface1* since their functionality, plane, and roles match, and therefore, the starting interface for *flow1* is VNF1-*Interface1*. The starting interface for *flow2* is VNF2-*Interface1* since it matches the *SAP2*.

The sequence of *flow1* starts from VNF1-*Interface1*. VNF1's *Flow Transformation*s matching this *Interface* for this *flow* are number *1* and *2*. Therefore, the next *exit interface* is VNF1-*Interface2* with functionality F6, control plane, and client and output roles. We did not select the *Flow Transformation3* since the role of VNF1-*Interface1* in it is output, but the roles of VNF1-*Interface1* as the starting interface are server and input. The next *entry interface* based on the dependency related to VNF1-*Interface2* is the VNF3-*Interface1* with functionality *F4*, control plane, and server and input roles. The roles of this interface are complementary roles of the previous interface, i.e. client and output. For the next *exit interface* in VNF3, we select the *Flow Transformation*s*1* and *2*, as they match the *current interface*'s characteristics (*F4*, control plane, server and input roles). Accordingly, the next *exit interface* is VNF3-*Interface1* with the role of output. At this point, the *current interface* and the next *exit interface* are the same, and there are more than one dependencies associated with this interface. Therefore, we are facing the aforementioned special case for finding the next *entry interface*. This interface has the server

role on the entry and the output role on the exit. Therefore, for finding the next *entry interface* we should use the same dependency as the one used in the incoming direction towards VNF3-*Interface1*. From this point, the flow is the response going back towards the same path as incoming. The next *entry interface* is VNF1-*Interface2* with the role of input. For finding the next *exit interface*, we select the *Flow Transformation3* in VNF1. The next *exit interface* is VNF1-*Interface1* with the output role. This is the end of the flow as it reaches the *SAP*.

We have designed the *Propagation Flow2* in a similar way. Table 4.1 shows the results of designing both *Propagation Flow*s in this example in details.

Interface Sequence / Propagation Flow	1 (Starting Interface)	2	3	4	5	6
1	VNF1-Interface1	VNF1-Interface2	VNF3-Interface1	VNF3-Interface1	VNF1-Interface2	VNF1-Interface1
	Func.: F5 **Plane:** Control **Roles:** Server, Input	**Func.:** F6 **Plane:** Control **Roles:** Client, Output	**Func.:** F4 **Plane:** Control **Roles:** Server, Input	**Func.:** F4 **Plane:** Control **Roles:** Output	**Func.:** F6 **Plane:** Control **Roles:** Input	**Func.:** F5 **Plane:** Control **Roles:** Output
2	VNF2-Interface1	VNF2-Interface2	VNF3-Interface1	VNF3-Interface1	VNF2-Interface2	VNF2-Interface1
	Func.: F7 **Plane:** Control **Roles:** Server, Input	**Func.:** F8 **Plane:** Control **Roles:** Client, Output	**Func.:** F4 **Plane:** Control **Roles:** Server, Input	**Func.:** F4 **Plane:** Control **Roles:** Output	**Func.:** F8 **Plane:** Control **Roles:** Input	**Func.:** F7 **Plane:** Control **Roles:** Output

Table 4.1 - Propagation Flow design example

4.2.7 Update of the Network Function Ontology

We can update the *NFO* by the previous successful experiences or based on new standards and architectures in the telecom domain. We update the *NFO* within or outside of the *NSD* generation process. Within the process, we update the *NFO* at Step 6 of the *NSD* generation process under specific conditions. Updating the *NFO,* in this case, is done automatically. Outside of the process, we may update the *NFO* automatically, or an expert may do it manually depending on different conditions.

4.2.7.1 *Update within the NSD generation process (Step 6)*

In this step, we update the *NFO* according to the new information that a tenant proposes in the *NSReq*. An '*unmatched*' element in the *SM* does not exist in the *NFO*. Therefore, the *NSReq* has new information when we have '*unmatched*' elements in the *SM*. The main condition for updating the *NFO* is the successful generation of at least one generic *NSD* at Step 5. This implies that the new information in the *NSReq* is reusable. Therefore, we can update the *NFO* accordingly. In this step, we update the *NFO* in two cases:

- **Case 1:** In this case, a functionality named A and its children in the *SM* are tagged as '*matched*' or '*from ontology*', and some of its *ComposedOf* associations are tagged as '*unmatched*'. It implies that all these functionalities (the functionality A and its children) exist in the *NFO*. However, the tenant has required functionalities as part of the functionality A's decomposition which is not in its decomposition in the *NFO*. This is new information on functionality A that the tenant has proposed. Two subcases may happen in which we may update the *NFO*.
    - **Case 1.1:** In this case, the whole core decomposition of the functionality A in the *NFO* exists in the *SM* with '*matched*' tag. It implies that the tenant has required the core decomposition of functionality A, in addition to new functionalities in A's decomposition. We take the updating action as discussed below.
    - **Case 1.2:** In this case, a part of the core decomposition of the functionality A in the *NFO* is tagged as '*from ontology*', not 'matched', in the *SM*. It implies that the tenant has not required the whole core decomposition of functionality A. In such a case, we consider the possibility of the tenant's mistake in naming the functional requirement A or its children in the *NSReq*. Therefore, before updating, we ask the tenant to validate the *NSReq*. In this way, we ensure that we do

not update the *NFO* based on a faulty *NSReq*. After the validation, we take the update action as discussed below.

Update action: The tenant cannot alter the core decomposition of a functionality in the *NFO*, as this information is established in the *NFO*. Therefore, we only add the required additional children for functionality A as its optional children in the *NFO*. To do so, we add the '*unmatched*' *ComposedOf* associations of functionality A in the *SM* to the *NFO*. Then, we add the dependencies of these additional children from the *SM* to the *NFO*, if they do not already exist there. These dependencies are only defined in functionality A's decomposition. Therefore, we set their contexts as functionality A in the *NFO*.

Figure 4.6 shows an example of *NFO* at the top left, and three different *SM*s originated from three different *NSReq* models at the right and the bottom. All three *SM*s fit in Case 1 of the *NFO* update, since *A*, *B*, *C*, *Y* and *Z* functionalities are *'matched'*. The core decomposition of *A* in the *NFO* (*B* and *C*) exists in *SM1*. Therefore, there is no possibility of the tenant's mistake in this example, and it fits Case 1.1. In *SM2* and *SM3*, however, *A*'s core decomposition does not exist exactly as it is in the *NFO*. Therefore, before the updating action, we ask the tenant to validate the *NSReq*. The updating action in all three cases is adding *Y* as an optional child for the *A*. In *SM1* and *SM2*, we add the *Y*'s dependency to the *NFO* with the context of functionality *A*. In *SM3*, we also add *Z* as the optional child of functionality *A*. Then, we add its dependency to *Y* with the context of functionality *A*.

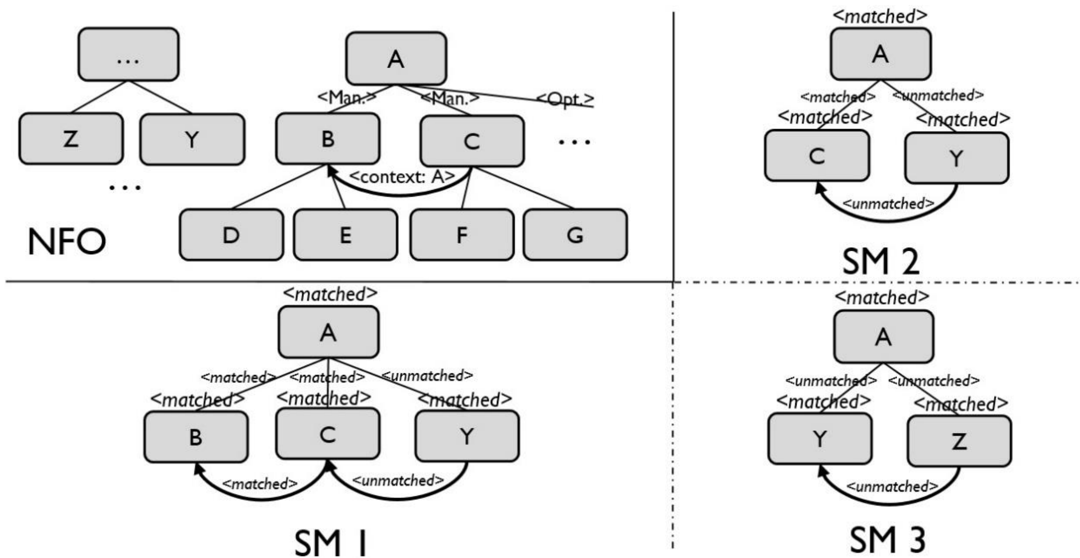

Figure 4.6 - NF Ontology updating case 1 example

- **Case 2:** In this case, a functionality named X is *'unmatched'* but all of its children are *'matched'* in the *SM*. Since functionality X is *'unmatched'* all of its *ComposedOf* associations are *'unmatched'* as well. Two subcases may happen with different update actions.

  - **Case 2.1:** In this subcase, the decomposition of functionality X does not exactly match any decomposition in the *NFO*. It implies that the tenant has proposed a new functionality with its decomposition in the *NSReq*.

    Update action: As the update action for Case 2.1, we create a new functionality in the *NFO* named X with the same decomposition as functionality X in the *SM*. Our assumption is that the whole decomposition that the tenant has proposed for functionality X is the core decomposition. Therefore, we define all of its *ComposedOf* associations as mandatory. We add all the dependencies defined between the children of functionality X into the *NFO*. We specify their contexts as functionality X.

- **Case 2.2:** In this subcase, the decomposition of functionality X is a subset of one of the functionalities' decompositions in the *NFO*. Also, it has the whole core decomposition of that functionality. This implies that functionality X's name is a new name (alias) for that functionality in the *NFO*.

  Update action: As the update action for Case 2.2, we add functionality X's name to the aliases of that functionality in the *NFO*.

Figure 4.7 shows four different *SM* examples. We consider the same *NFO* as shown in Figure 4.6 as the *NFO* for these examples. These four *SM* instances fit in Case 2, as functionality *X* is '*unmatched*', and its children in all the examples are '*matched*' including *B*, *C*, *Y*, and *Z*. The first three *SM* instances fit into case 2.1. Therefore, for each of these *SM*s, we add the functionality *X* into the *NFO* with the same decomposition and dependencies. *SM4* fits into subcase 2.2, as its decomposition is exactly the same as functionality *A*'s core decomposition in the *NFO*. In this case, we add *X* as an alias for functionality *A* in the *NFO*.

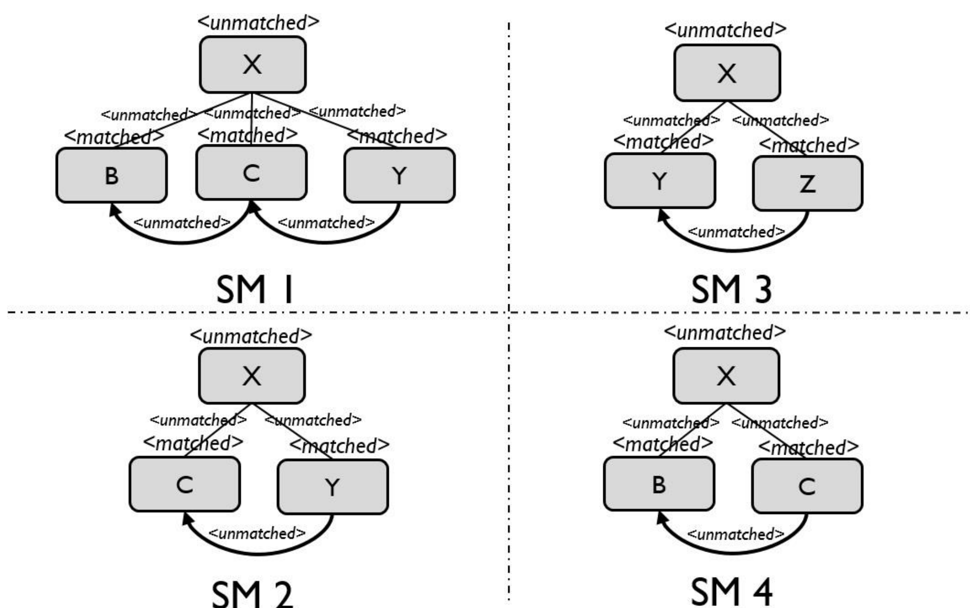

Figure 4.7 - NF Ontology updating case 2 example

4.2.7.2 *Update Independent of the NSD generation process*

This kind of *NFO* update may take place anytime independently of the *NSD* generation process. It consists of two different cases:

Case 1 (Automatic): This case of *NFO* update is done automatically as soon as a VNF, i.e. a *VNF Package Info* model, is added to the *VNF Catalog*. First, we check whether there are any architectural blocks in the *NFO* that match this VNF. If not, we create a functionality element and an architectural block element realizing that functionality, both with the same name as the VNF's '*implementedArchBlock'* attribute. This way we can reuse this VNF to compose network services using the *NFO*. For each of the VNF's interfaces – defined in its *VNFAD* – we create an *Interface* element and its related *Interface Functional Characteristic* elements for the created architectural block.

- **Case 2 (Manual):** This case of *NFO* update is done manually by an expert. The expert can add new functionalities, architectural blocks and their interfaces, decompositions, and dependencies, and *SAP*s with their related elements. In addition, the expert may modify the existing elements and their attributes in the *NFO* as well. The expert does the modifications manually through an interface depending on how the models are implemented. For instance, if the models are UML files, they can be modified using UML editors.

4.3 Limitations

Our proposed method does not support designing multi-domain network services. For example, we cannot generate an *NSD* for an IMS service with home and visited network domains. We are only able to design each domain in a separate *NSD*. A multiple-domain network service requires having multiple VNF instances of the same *VNFD* with different roles, e.g. two P-

CSCF VNFs (same type), one for the home and the other for the visited network domain. This is not supported by our method.

Our method does not validate the consistency of the *NSReq*. We assume the *NSReq* is consistent and has no conflicting requirements. If there are conflicting requirements in an *NSReq*, there is no guarantee about the validity of the network services generated from such requirements.

Our method is limited to using only VNFs as network functions in the network service. PNFs and nested network services are not taken into account for the design of the network service.

There are three flow patterns for the virtual links including *E-Line, E-LAN,* and *E-Tree* [28], as discussed in Section 2.2.1. Our method does not support the *E-Tree* flow pattern for the virtual links. To include it specific information should be provided by the architectural dependencies for the *E-Tree* flow pattern.

Chapter 5

Network Service Descriptor Refinement (w.r.t Non-functional Requirements)

In this chapter, we describe our approach for tailoring a generic *NSD* based on the *NFR*s included in the *NSReq*. The approach enriches the generic *NSD*'s deployment flavor and its elements. As a result, the network service will be capable of providing the required QoS range, from the minimum to the maximum as indicated by the *NFR*s.

5.1 Overall approach

The *NFR*s indicate QoS requirements only for the functionalities exposed by the *SAP*s, as discussed in Section 3.1.4. However, we need to dimension the whole network service. Therefore, we propagate the *NFR*s in the network service using the *Propagation Flow*s that we generated in Section 4.2.5.5. By propagating the *NFR*s, we calculate the QoS requirements for all the VNFs. We dimension the VNFs based on their QoS requirements. To tailor a generic *NSD*, we need to tailor its VNFs by dimensioning them, and selecting a deployment flavor for each, and calculate the virtual links' capacity. For each VNF, we select a deployment flavor according to the VNF's dimensioning and some other criteria that we define. We calculate the capacity of

the virtual links and tailor their deployment flavors according to the VNFs' dimensioning. Eventually, we tailor the network service deployment flavor according to the VNFs' and the virtual links' deployment flavors.

This process takes a generic *NSD* and its corresponding *SM* model as inputs. We need the *SM* model, as it has the *NFR*s and architectural information of the network service. The approach consists of three steps:

1- Propagating the *NFR*s
2- Dimensioning the VNFs
3- Tailoring the *NS Deployment Flavor*s

In the following section, we will discuss the details of each step.

5.2 Steps of the Approach

5.2.1 Propagating the NFRs

In this step, we propagate the *NFR*s through the *Propagation Flow*s designed for the network service. As discussed in Section 3.3.2.1, each VNF interface exposes a specific QoS for all of its exposed functionalities to outside. A *Propagation Flow* consists of VNF interfaces. By propagating the *NFR*s through each *flow*, we calculate the required QoS from each interface involved in the *flow*. According to the QoS requirements (*QR*) for each interface, we calculate its total instance utilization (IU), and this is the goal of this step.

In each *flow,* we calculate the *QR*s for each appearance of an interface (*SmInterface*) in the *flow*'s sequence. A *QR* for an *SmInterface* is a transformation of one of the *QR*s of the previous *SmInterface* in the *flow*. We start these transformations from the *NFRs* related to the *flow*. Originally, each *flow* is designed based on an *Exposed Functionality* element of a *SAP*. Each *NFR* associates with an *Exposed Functionality* of a *SAPR* as well. Therefore, each *NFR* relates to a

flow based on this association. In a *QR* transformation, we may change its value and/or metric for creating a new *QR*. For each new *QR*, we create a *QR* element in the *SM* associated with its *SmInterface,* and we reference its source, i.e. a *Flow Transformation* element or an *NFR*. Depending on the *SmInterface* being an *entry* or *exit interface* the procedure of calculating its *QR*s differs.

5.2.1.1 *Calculating the QRs for an entry interface*

In a *flow*'s sequence, an *entry interface* and its previous *exit interface* are related through a dependency, as discussed in Section 4.2.5.5. The starting interface of the *flow* is an exception, as it is directly connected to the *SAP*. In the network service, there is only a virtual link between an *entry interface* and its previous *exit interface*. As discussed in Section 3.7.1, the QoS characteristics of virtual links like delay and jitter are out of our scope, and we neglect them in dimensioning the network service. Therefore, nothing changes the QoS characteristics of the packet flows between such interfaces. As a result, the *QR*s of each *entry interface* is exactly the same as the *QR*s of its previous *exit interface* in the sequence.

The *QR*s of the *flow*'s starting interface are the ones defined by the *NFR*s related to the *flow*. As discussed in Section 3.1.4, the composite *NFR*s have an extra *QR* of throughput as well as their RPS *QR*. Its value is the multiplication of the *NFR*'s *'value'* by the *NFR*'s *'request size'*.

5.2.1.2 *Calculating the QRs for an exit interface*

In designing a *flow*, we discover each *exit interface* based on one or multiple *Flow Transformations* related to the previous *entry interface*, as discussed in Section 4.2.5.5. Such *Flow Transformation*s specify the transformation of QoS characteristics of packet flows between these two interfaces by *QoS Ratio* elements. Therefore, we derive the *QR*s for each *exit interface* by transforming the *QR*s of the previous *entry interface* according to the *QoS Ratio* elements of those *Flow Transformation*s.

Among these *QoS Ratio* elements, only some are related to each *QR* of the previous *entry interface*. If the metric that a *QoS Ratio* specifies for the *entry interface* is the same as the *QR*'s metric, the *QoS Ratio* is related to the *QR*. We transform each *QR* of the previous *entry interface* using each of its related *QoS Ratio*s. Therefore, we generate one *QR* for each related *QoS Ratio*. The metric of the new *QR* is the same as the metric that the *QoS Ratio* defines for the *exit interface*. The value of the new *QR* is equal to the multiplication of the previous *QR*'s value by the *QoS Ratio*'s ratio. If the flow is backward propagation, we multiply the value by 1/ratio, as we are propagating in the opposite direction of the *Flow Transformation*.

5.2.1.3 *Calculating the total IU for the interfaces*

A *QR* defined for an *SmInterface* specifies a load on the interface. Therefore, each *QR* results in an instance utilization for its interface, and we call it metric.IU. We calculate a metric.IU by dividing the *QR*'s value by the value of the interface's *QoS Characteristic* with the same metric as the *QR*. The *QR* is not relevant to its interface if the interface has no *QoS Characteristic* with the same metric as the *QR*. Therefore, we dismiss such *QR*.

Different *QR*s for an *SmInterface* describe the same load with different metrics on the interface in one of its appearances in the flow. The maximum among the metric.IUs of an *SmInterface* is the instance utilization for that *SmInterface* (i.e. a specific appearance of the interface in the *flow*). The total load on an interface in the whole network service (i.e. the load from all the *flows* involving the interface) is equal to the summation of the loads of all of its appearances in all *flow*s. Therefore, the total instance utilization of an interface is equal to the summation of the maximum metric.IU of all of its related *SmInterface*s in all *flow*s. We refer to it as the total.IU. We calculate the total.IU for all the interfaces involved in at least one *flow* in the network service.

Table 5.1 shows the *QR*s and the metric.IUs for each appearance of the interfaces in the *flow*s in the example discussed in Section 4.2.6. According to this table, the total.IU for VNF1-*Interface1*, VNF1-*Interface2*, VNF2-*Interface1*, VNF2-*Interface2*, and VNF3-*Interface1* are equal to 9.2, 8, 15.25, 11.5 and 6.8, respectively.

Propagation Flow / Interface Sequence	1 (Starting Interface)	2 (exit)	3 (entry)	4 (exit)	5 (entry)	6 (exit)
1	VNF1-Interface1	VNF1-Interface2	VNF3-Interface1	VNF3-Interface1	VNF1-Interface2	VNF1-Interface1
	QR1: Throughput – 600	**QR1:** Throughput – 720 (by Flow Trans. 2)	**QR1:** Throughput – 720	**QR1:** Throughput – 2160 (by Flow Trans. 2)	**QR1:** Throughput – 2160	**QR1:** Throughput – 2160 (by Flow Trans. 3)
	metric.IU 1: 600/300 = 2	**metric.IU 1:** 720/360 = 2	**metric.IU 1:** 720/600 = 1.2	**metric.IU 1:** 2160/600 = 3.6	**metric.IU 1:** 2160/360 = 6	**metric.IU 1:** 2160/300 = 7.2
2	VNF2-Interface1	VNF2-Interface2	VNF3-Interface1	VNF3-Interface1	VNF2-Interface2	VNF2-Interface1
	QR1: RPS – 20 **QR2:** Throughput – 200	**QR1:** RPS – 20 (by Flow Trans. 1) **QR2:** Throughput – 300 (by Flow Trans. 2)	**QR1:** RPS – 20 **QR2:** Throughput – 300	**QR1:** Throughput – 900 (by Flow Trans. 1) **QR2:** Throughput – 900 (by Flow Trans. 2)	**QR1:** Throughput – 900 **QR2:** Throughput – 900	**QR1:** Throughput – 900 (by Flow Trans. 3) **QR2:** Throughput – 900 (by Flow Trans. 3)
	metric.IU 1: 20/5 = 4 **metric.IU 2:** 200/80 = **2.5**	**metric.IU 1:** 20/5 = 4 **metric.IU 2:** 300/120 = **2.5**	**metric.IU 1:** 20/40 = 0.5 **metric.IU 2:** 300/600 = **0.5**	**metric.IU 1:** 900/600 = 1.5 **metric.IU 2:** 900/600 = **1.5**	**metric.IU 1:** 900/120 = 7.5 **metric.IU 2:** 900/120 = **7.5**	**metric.IU 1:** 900/80 = 11.25 **metric.IU 2:** 900/80 = **11.25**

Table 5.1 - NFR propagation example

5.2.2 Dimensioning the VNFs

In this step, we dimension all the VNFs in the network service according to the instance utilization of their interfaces. As discussed in Section 3.3.2.1, a VNF interface exposes a portion of its related VNFC's QoS capacity. Therefore, we can dimension a VNFC according to the instance utilization of its interfaces. The QoS capacity of a VNFC is, in fact, a portion of the VNF's QoS capacity. Therefore, by dimensioning the VNFCs of a VNF we can select a suitable deployment flavor for the VNF, and dimension it.

5.2.2.1 *Dimensioning a VNFC*

To dimension a VNFC we calculate its required number of instances, and we call it VNFC.RI. As discussed in Section 3.3.2.1, we assume that the VNFC's QoS capacity is compartmentalized between its VNF interfaces. Therefore, the capacity portion accessed through an interface is separate from the others. This means one instance of a VNFC can provide the QoS exposed by all of its interfaces simultaneously. If the number of instances of a VNFC is equal to the maximum of the instance utilization of its interfaces, it is certain that the *QR*s of all of its interfaces are fulfilled. Therefore, the VNFC.RI is equal to the ceiling of the maximum total.IU of the VNFC's interfaces, as shown in Equation 1.

$$VNFC.RI = \left\lceil max\left\{i \in VNFC^{\prime}\ s\ interfaces\ involved\ in\ at\ least\ a\ flow\ \middle|\ total.IU_i\right\}\right\rceil \quad (1)$$

In the example discussed in Section 4.2.6, in VNF1, VNFC1 has the *interface1*, and VNFC2 has the *interface2*. The total.IU for interface 1 and 2 are 9.2 and 8 respectively, as discussed in the previous step. Therefore, VNFC.RI for the VNFC1 and 2 are 10 and 8 respectively. For VNF2, in a similar way, interface 1 and 2 have the total.IUs of 15.25 and 11.5, therefore VNFC.RI for VNFC1 and 2 are 16 and 12, respectively. In VNF3, the total.IU for the interface1 is 6.8, therefore the VNFC.RI for its VNFC1 is 7.

5.2.2.2 *Dimensioning a VNF*

As discussed before, each *Instantiation Level* of a VNF specifies the number of instances of each VNFC in the VNF at the instantiation time. In order to fulfill the required number of instances of VNFCs of a VNF, we can use any of the VNF's *Instantiation Level*s with a specific number of VNF instances. We consider each of these combinations as a solution for dimensioning the VNF. In each solution, we refer to the required number of instances as $VNF.RI_{IL}$. To dimension a VNF, first, we calculate the $VNF.RI_{IL}$ for all the *Instantiation Level*s, i.e. calculate

all the solutions. Depending on specific criteria that we will define later one of these solutions is more suitable than the rest. Based on the criteria we select one solution in each *VnfDf*, and then we select one *VnfDf* as the final solution.

Number of VNF instances for an Instantiation Level: For an *Instantiation Level*, we compare the required number of instances for each VNFC with its number of instances that the level specifies in the related *Vdu Level*. Accordingly, we calculate the required number of the VNF instances for the *Instantiation Level* only based on the VNFC. We refer to it as the $VNF.RI_{VNFC}$. The $VNF.RI_{VNFC}$ is equal to the ceiling of the division of the VNFC.RI by the *Vdu Level*, as shown in Equation 2.

Each $VNF.RI_{VNFC}$ fulfills the required number of instances of that specific VNFC in the *Instantiation Level*. Therefore, the maximum of all the $VNF.RI_{VNFC}$s fulfills the required number of instances for all the VNF's dimensioned VNFCs in the *Instantiation Level*. That is the required number of instances of the VNF in the *Instantiation Level*, i.e. $VNF.RI_{IL}$, as shown in Equation 3.

$$VNF.RI_{VNFC} = \left\lceil \frac{VNFC.RI}{Vdu\ level} \right\rceil \quad (2)$$

$$VNF.RI_{IL} = max\{i \in all\ of\ the\ VNF'sdimensioned\ VNFCs \mid VNF.RI_{VNFCi}\} \quad (3)$$

Selecting the desirable solution: After having all the solutions, we select one of them, i.e. an *Instantiation Level* with its $VNF.RI_{IL}$, according to the criteria we define. Our criteria include '*Flexible scaling'* and '*VNFCs' failure impact'*. The VNFs with more flexibility in scaling and less impacted by their VNFCs' failure are desirable. We prioritize the first criterion.

Flexible scaling criterion: In the NFV framework today, all instances of a VNF in a given role are instantiated using the same *VnfDf* and *Instantiation Level*. The traffic among these instances is typically load-balanced. Therefore, all these instances should have the same capacity, and if we scale one we should scale all other instances as well. As a result, increasing the number of VNF instances decreases the scaling flexibility, since its granularity of scaling decreases. For instance, scaling a VNF instance to the next step results in adding 3 VNFC instances. If we have two instances for the VNF, scaling it to the next step adds a total number of 6 VNFC instances, and it is less granular. Hence, according to this criterion, we prefer a VNF with smaller VNF.RI. This results in selecting deployment flavors that provide more capacity for a single instance.

VNFC's failure impact criterion: The failure impact of a VNFC is related to its capacity. The less capacity a VNFC has, the less impact its failure has on the VNF and the network service. Therefore, to dimension a VNF the solution with the least capacity for the VNFCs is desirable according to this criterion.

Instantiation Level selection in a VnfDf: As discussed earlier, we prioritize the first criterion over the second one. Therefore, in each *VnfDf*, we select the *Instantiation Level* with the minimum $VNF.RI_{IL}$ as the desirable solution. If there are multiple *Instantiation Level*s with the minimum $VNF.RI_{IL}$ in the same *VnfDf*, the ones with a bigger total number of VNFC instances provide unnecessary resources. Thus, we select the one with the least total number of VNFC instances as the solution in the *VnfDf*. The selected *Instantiation Level* in each *VnfDf* is the largest level that fulfills the *NFR*s by providing the least unnecessary resources.

VnfDf selection: Among the *VnfDf*s we select one as the final solution for dimensioning the VNF. For this selection, we consider the first criterion. Therefore, we select the *VnfDf* that its selected *Instantiation Level* has the minimum $VNF.RI_{IL}$. If there are multiple

*VnfDf*s with the minimum VNF.RI_{IL} for their selected *Instantiation Level*s, we consider the second criterion for selection. The selected solutions in each *VnfDf* has the minimum extra VNFC instances, and they all fulfill the *NFR*s. Therefore, all these solutions provide approximately the same QoS capacity. Thus, the solution with the more total number of VNFC instances has VNFCs with less capacity. According to the second criterion, we select the solution with the most total number of VNFC instances.

Dimensioning the VNFs without QoS requirement: As discussed in Section 4.2.5.5, we design the *flow*s based on the *SAP*s, and we define the *SAP*s according to the *SAPR*s which the tenant defines. It implies that the tenant has the knowledge about the packet flows in the network service to some extent. We assume that the tenant defines the *SAPR*s and *NFR*s in a way that all the VNFs are involved in at least one *flow*, and they are dimensioned to this point. If a VNF is not dimensioned, it implies that the VNF is not involved in any *flow* and/or there is no QoS requirement for it. We dimension each of these VNFs to its default values, i.e. we select its default *VnfDf* and *Instantiation Level* with the VNF.RI equal to 1.

Table 5.2 shows the *VnfDf*s of the VNFs in the example of Section 4.2.6 and the results of dimensioning them. Each row of the table for a VNF shows a solution for dimensioning the VNF. The solutions with a solid or dashed circle are the candidate solutions in each *VnfDf*. The ones with a solid circle are the final solutions for dimensioning each VNF.

VNFs / VnfDfs		VNF1				VNF1 Dimensioning		VNF2				VNF2 Dimensioning		VNF3		VNF3 Dimensioning	
		VNFC1 (RI=10)	VNF. RI_{VNFC}	VNFC2 (RI=8)	VNF. RI_{VNFC}	VNF.RI	Total VNFCs	VNFC1 (RI=16)	VNF. RI_{VNFC}	VNFC2 (RI=12)	VNF. RI_{VNFC}	VNF.RI	Total VNFCs	VNFC1 (RI=7)	VNF. RI_{VNFC}	VNF.RI	Total VNFCs
Df 1	Ins. Lvl. 1	1	$\lceil 10/1 \rceil = \mathbf{10}$	1	$\lceil 8/1 \rceil = \mathbf{8}$	**10**	20	1	$\lceil 16/1 \rceil = \mathbf{16}$	1	$\lceil 12/1 \rceil = \mathbf{12}$	**16**	32	1	$\lceil 7/1 \rceil = \mathbf{7}$	7	7
	Ins. Lvl. 2	3	$\lceil 10/3 \rceil = \mathbf{4}$	3	$\lceil 8/3 \rceil = \mathbf{4}$	**4**	24	3	$\lceil 16/3 \rceil = \mathbf{6}$	3	$\lceil 12/3 \rceil = \mathbf{4}$	**6**	36	2	$\lceil 7/2 \rceil = \mathbf{4}$	**4**	8
	Ins. Lvl. 3	5	$\lceil 10/5 \rceil = \mathbf{2}$	5	$\lceil 8/5 \rceil = \mathbf{2}$	**2**	20	5	$\lceil 16/5 \rceil = \mathbf{4}$	5	$\lceil 12/5 \rceil = \mathbf{3}$	**4**	40	3	$\lceil 7/3 \rceil = \mathbf{3}$	**3**	9
Df 2	Ins. Lvl. 1	7	$\lceil 10/7 \rceil = \mathbf{2}$	7	$\lceil 8/7 \rceil = \mathbf{2}$	**2**	28	7	$\lceil 16/7 \rceil = \mathbf{3}$	7	$\lceil 12/7 \rceil = \mathbf{2}$	**3**	42	4	$\lceil 7/4 \rceil = \mathbf{2}$	**2**	8
	Ins. Lvl. 2	9	$\lceil 10/9 \rceil = \mathbf{2}$	9	$\lceil 8/9 \rceil = \mathbf{1}$	**2**	36	9	$\lceil 16/9 \rceil = \mathbf{2}$	9	$\lceil 12/9 \rceil = \mathbf{2}$	**2**	36	5	$\lceil 7/5 \rceil = \mathbf{2}$	**2**	10
	Ins. Lvl. 3	11	$\lceil 10/11 \rceil = \mathbf{1}$	11	$\lceil 8/11 \rceil = \mathbf{1}$	**1**	22	11	$\lceil 16/11 \rceil = \mathbf{2}$	11	$\lceil 12/11 \rceil = \mathbf{2}$	**2**	44	6	$\lceil 7/6 \rceil = \mathbf{2}$	**2**	12

Table 5.2 - VNF dimensioning example

5.2.3 Tailoring the NS Deployment Flavor

In this step, we tailor the *NsDf* of the generic *NSD*, according to the *NFR*s. In order to tailor the *NsDf*, we tailor the *VNF* and *NsVL Profile*s, and we generate all the *NS Level*s according to the dimensioning of the VNFs.

5.2.3.1 *Tailoring the VNF Profiles*

The *VNF Profile*'s attributes that are in our scope include the reference to the *VnfDf* and the *Instantiation Level*, and the minimum and the maximum number of instances for the VNF. NFV-MANO instantiates each VNF based on its referenced *Instantiation Level*. We want to instantiate the VNFs with their average *Instantiation Level*. To select the average *Instantiation Level* for each VNF, we sort all of its *Instantiation Level*s. The *Instantiation Level* in the middle of the minimum and the selected *Instantiation Level* is the average, and we reference it in the *VNF Profile*. We set the minimum number of instances for each VNF to 1, as a VNF cannot have less

than 1 instance in the network service. We set the maximum to the VNF.RI_{IL} of the selected solution, as it specifies the maximum capacity a VNF can have to fulfill the *NFR*s.

5.2.3.2 *Tailoring the NsVl Profiles*

In order to tailor the *NsVl Profile*'s, we should calculate the minimum and the maximum bitrate requirements of the virtual link.

Virtual link bitrate requirement: The connection of a VNF instance to a virtual link is through a VnfExtCp as discussed earlier. The VnfExtCp is also connected to one or many instances of a VNFC through their VnfcCps. Different instances of the same VNFC type have the same QoS characteristics including their VnfcCps bitrate requirement. Therefore, the bitrate requirement of a virtual link's connection is equal to the VnfcCp bitrate requirement multiplied by the number of VNFC instances. The bitrate requirement of a VnfcCp is provided in its descriptor (*VduCpd*) in the VNFD, as shown in VNFD metamodel.

The bitrate requirement of a virtual link has two attributes including the root and the leaf bitrate requirements, and both are numerical. The root and the leaf bitrate requirements for different virtual link types are different, and [9] has defined them – in this work we only generate E-Line and E-LAN virtual links, therefore we will not discuss E-Tree bitrate requirement.

E-Line virtual link bitrate requirement: As discussed in Section 2.2.1, an *E-Line* virtual link has only two connections. Thus, the number of instances of both VNFs connected to it can be only one. The root bitrate requirement of an *E-Line* virtual link is equal to the bitrate of the line. The line bitrate is the summation of the bitrate requirements of the two connections. Equation 4 shows the root bitrate requirement calculation. The leaf bitrate requirement is not applicable for the *E-Line* virtual links.

$$E - Line\ Root\ Bitrate\ Requirement = \sum_{i \in both\ of\ connected\ VNF\ types} Vdul\ Level_i * VduCp\ Bitrate\ Requirement_i \quad (4)$$

E-LAN virtual link bitrate requirement: An *E-LAN* virtual link has more than two connections to the VNFs. Each connection is a leaf, and the root is the aggregate capacity of the LAN. The root bitrate requirement is equal to the summation of the bitrate requirements of all the leaves. Equation 5 shows the root bitrate requirement calculation in which K is the number of instances for each VNF type connected to the *E-LAN*.

$$E - LAN\ Root\ Bitrate\ Requirement = \sum_{i \in all\ of\ connected\ VNF\ types} K_i * Vdul\ Level_i * VduCp\ Bitrate\ Requirement_i \quad (5)$$

The NFV framework at this point does not support different leaf bitrate requirements for a virtual link [9]. Therefore, the leaf bitrate requirement of an *E-LAN* is equal to the maximum bitrate requirement among all of its connections. Equation 6 shows the *E-LAN* leaf bitrate requirement calculation.

$$E - LAN\ Leaf\ Bitrate\ Requirement = max\{i \in all\ of\ connected\ VNF\ types \mid Vdu\ Level_i * VduCp\ Bitrate\ Requirement_i\} \quad (6)$$

A virtual link has the minimum bitrate requirement if each of its connected VNFs has 1 instance, and they are instantiated using their smallest *Instantiation Level*s. Similarly, it has the maximum bitrate requirement if each connected VNF has the number of instances equal to its $VNF.RI_{IL}$ and they are instantiated using their selected *Instantiation Level*. To calculate the minimum and maximum bitrate requirements, we consider the '*Vdu Level*' parameter in the equations according to the minimum and the selected *Instantiation Level*s, respectively. The number of VNF instances only affects the *E-LAN* root bitrate requirement, i.e. 'K' parameter in the equation. For the minimum and the maximum bitrate requirements, we consider 'K' equal to 1 and the $VNF.RI_{IL}$ for each VNF type, respectively.

5.2.3.3 *NS Levels Generation*

We generate all the possible *NS Level*s according to the VNFs dimensioning, so the network service can be scaled in the range of the required QoS. The network service provides the minimum and the maximum QoS when each VNF has the number of instances equal to 1 and the $VNF.RI_{IL}$, respectively. We create an *NS Level* for the *NsDf* for each combination of the VNFs' number of instances in this range.

The main elements of an *NS Level* are *VnfToLevelMapping* and *VirtualLinkToLevelMapping*, as discussed in Section 3.7.4. For each *NS Level*, we create one *VnfToLevelMapping* for each VNF and one *VirtualLinkToLevelMapping* for each virtual link. For each *VnfToLevelMapping*, we set the number of instances equal to the VNF's number of instances in that *NS Level*, and we reference the *VNF Profile* of the VNF. For each *VirtualLinkToLevelMapping*, we reference the *NsVl Profile* of the virtual link, and we create a *LinkBitrateRequirement* element.

We calculate a virtual link's bitrate requirement in an *NS Level* using the same equations as the bitrate requirement of its *NsVl Profile*. As discussed earlier, each *NS Level* specifies the VNFs' number of instances. Therefore, the 'K' parameter for the *E-LAN* root bitrate requirement (Equation 5) is equal to the number of instances defined by *NS Level*.

5.3 Limitations

In our approach, we generate the *NS Level*s exhaustively and not based on the scaling policies. Therefore, some of them are not useful at runtime. According to the scaling rules, only the ones that are used for scaling the network service should remain in the *NSD*.

Our approach does not target the best *NSD* among the generated ones. All the generated *NSD*s fulfill the functional, architectural, and non-functional requirements. However, their non-functional characteristics are different, and some may be more efficient than others.

Chapter 6

Prototype Tool

We have developed a prototype tool for implementing our approach. In this chapter, we will discuss the architecture of the prototype we developed and its application to a case study. We developed the prototype using a model-driven approach. As model transformation language, we selected ATL [36]. ATL is simple, and it is flexible with both imperative and declarative programming paradigms. We have developed the models and metamodels introduced in Chapter 3 using Papyrus tool [34].

6.1 Prototype architecture

Our approach captures the information from different input models into the *SM* model incrementally and manipulates it to fulfill its goal. Each input model in our approach conveys different information that affects the approach differently. In order to simplify our approach, we have separated it into different steps according to the input models, as discussed in Chapter 4 and 5. We have developed our prototype tool according to these steps to reduce its complexity and ease the debugging.

Our prototype tool consists of six ATL transformations, each transformation implements one of the steps of the approach. We execute these transformations sequentially. However, there

are two exceptions. Step 3, as discussed in Section 4.2.3, is a simple step. It gets the *SM* and *VNF Catalog* models as inputs and outputs the refined *SM*. Step 4, as discussed in Section 4.2.4, gets the *SM* model from Step 3 and *Protocol Stack* model as inputs. It only refines the *SM* model and outputs it. The difference between the inputs and outputs of these two steps is not significant. To simplify the implementation, we have combined these two steps in one transformation. Step 5 and 6, have the same inputs and different outputs, and these steps do not affect each other. Therefore, they can get executed in parallel. Figure 6.1 shows the flow diagram of the prototype with the inputs and outputs of the model transformations.

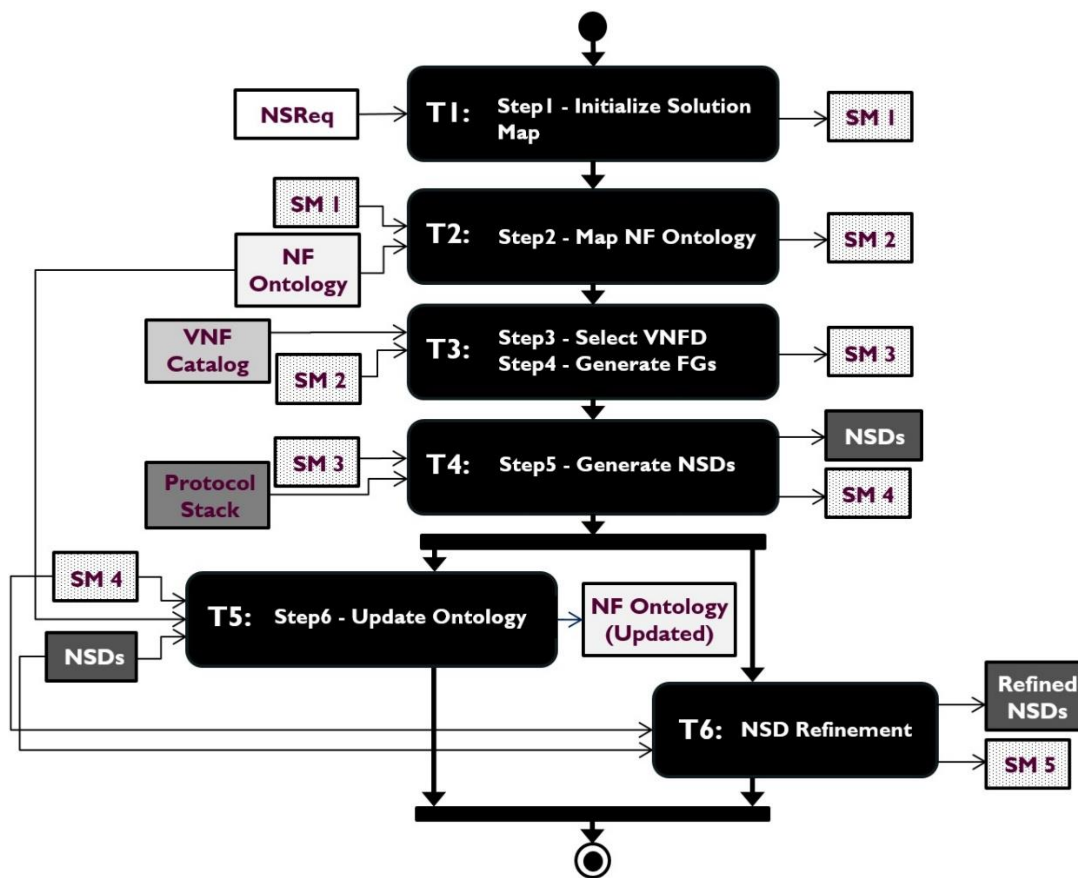

Figure 6.1 - A flowchart for the transformations in the prototype tool

6.1.1 Transformation structure

The nature of each step of our approach is sequential, i.e. handling the details in the right order is crucial. For instance, at Step 2, we traverse each functionality in the *SM* through its realization relations first. If we traverse it through its decomposition relations before the other relations it has a different outcome, and it is not desirable for us. Therefore, it is important to implement the transformations in an imperative manner. In the ATL language, a `lazy rule` is called from other rules [36]. Therefore, we can execute the `lazy rule`s sequentially. The `do` section of a rule is also the imperative block that we can use to call the `helper`s and `lazy rule`s in the desirable order.

In our prototype, all the transformations have a similar structure which is the result of the ATL language structure and limitations, as we will discuss in Section 6.1.2. In each transformation, we need to create the elements of the output model(s) in the right order. Therefore, we have defined a `lazy rule` for each of these elements. For instance, to create the *VNF Profiles* in an *NSD* model we have defined the `CreateVnfProfile lazy rule` in Transformation 4, as shown in Figure 6.2.

```
lazy rule CreateVnfProfile{
    from
        s : OclAny,
        packageName: String

    to
        tp: NSD!VnfProfile(
            base_Class <- t,
            vnfProfileId <- 'VnfProfile-'+s.base_Class.name
        ),

        t: NSD!Class (
            name <- 'VnfProfile - ' + s.base_Class.name,
            package <- thisModule.getPackageOUT(packageName)

        )
    do{
        tp.Vnfd.add(s);
    }
}
```

Figure 6.2 - An example of an ATL lazy rule in the prototype

For each function that can be implemented declaratively, we have defined a `helper`. For instance, in many transformations, we need to calculate the Cartesian product of two sequences that contain other sequences. We have defined the `multiplySeqs(Seq1, Seq2) helper` for this function, as shown in Figure 6.3.

```
--Multiplies two sequences of sequences.
helper def: multiplySeqs(Seq1: Sequence(Sequence(OclAny)), Seq2:
    Sequence(Sequence(OclAny))) : Sequence(Sequence(OclAny)) =

    if Seq1.isEmpty() then Seq2
        else if Seq2.isEmpty() then Seq1
            else
                Seq1->iterate(e; acc1 : Sequence(Sequence(OclAny)) =
                    Sequence{} | acc1.union(Seq2 ->
                        iterate(f; acc2 : Sequence(Sequence(OclAny)) =
                            Sequence{} | acc2->including(e.union(f))))
                    )
            endif
        endif;
```

Figure 6.3 - An example of an ATL helper in the prototype

```
rule Main {
    from
        s: SM!Model

    to
        t: SM!Model (
            name <- 'SM'

        ),

        t2: SM!Class(
            name <- 'VnfdOnboardedPackages',
            package <- t
        ),
        tp2: SM!VnfdOnboardedPackages(
            base_Class <- t2
        )


    do{
        t.applyProfile(SM!Profile.allInstancesFrom('SM')->first());

        -- ~~~~~~~~~~~~~~~~~~~~~~~~~~<<Map Functionality & ABs to VNF Catalogue>>~~~~~~~

        for(e in SM!Functionality.allInstancesFrom('OUT').union( SM!ArchBlock.allInstanc
            for(f in VNFCatalogue!VnfCatalogue.allInstancesFrom('IN1').first().VnfPackag

                if (f.userDefinedData->select(m | m.valueType = #REFERENCE and m.key = '
                        first().valueVnfArchitecturalDesc.functionality -> exists( z | S
                    ){
                    e.vnfd.add(f.vnfd);
                    thisModule.getOnboardedPackages().packages.add(f);
                    e.isVnfMapped <- true;
                    if ( e.oclType() = SM!ArchBlock ){
                        for(g in e.getABFsOUT()){
                            g.isVnfMapped <- true;
                        }
                    }
                }
```

Figure 6.4 - An example of a Main rule in the prototype

In each transformation, we have defined a `matched rule` that is called automatically at the beginning of the transformation, and we refer to it as the `Main rule`. We create the output model(s) using the `to` section of the `Main rule`, and we use its `do` section as the main function. We implement the algorithms of the approach's step(s) in this section by calling the `help-ers` and `lazy rules` imperatively, and using the basic imperative commands, e.g. `if`, and `for`. Figure 6.4 shows a portion of the `Main rule` of Transformation 3. Figure 6.5 shows an overview of the structure of our transformations.

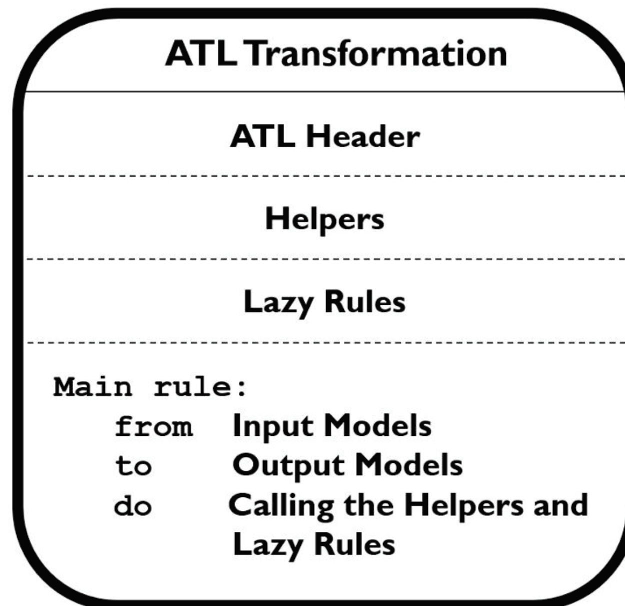

Figure 6.5 - An overview of the transformations' structure in the prototype

6.1.2 Challenges in using the ATL language

We have faced two major challenges with the usage of the ATL language.

1) The main focus of the ATL language is on the declarative programming paradigm. Many of the ATL features are not developed for imperative blocks. For instance, we can only define the ATL `helpers` declaratively. We cannot implement the imperative functions using the `helpers`. Therefore, we should implement them in the `do` section of the rules, and we cannot reuse them. In addition, defining local variables in the imperative blocks is not possible. Therefore, in the `do` sections, we are not able to store a value to reuse it. We should recalculate it each time we need it which is inefficient.
2) In each transformation, we refine the input *SM* model, and we output it. Therefore, the input *SM* should be copied in the output *SM*, and our refinement should be added to it. The ATL language does not handle such a case by default. It has a special mode called Refining mode in which it overwrites the changes in the input model. As of yet, the Refining mode does not support the imperative blocks of the ATL language, i.e. `do`

sections and `lazy rules`. Therefore, we cannot use this mode to create the output *SM*s. In each transformation, we create a new *SM* model. At the beginning of the transformation, we copy each element of the input *SM* in it, and then we manipulate it throughout the transformation. This solution is inefficient.

6.2 Case study: VoLTE service using IMS architecture

As a case study, we use the VoLTE [12] service using IMS [13] architecture. We use VoLTE since it is a popular service today. We also use the IMS since it is a complex architecture with many features and extensive documentation. By investigating the IMS we were inspired to add multiple features to our method, so it can handle such complexities. These features include the *Context* element and the characteristics of the *SAP* and *Interface* elements. It also helped us to grasp the possible complexities of the packet flows in the network services.

6.2.1 NSReq

In the *NSReq* of our case study, the tenant requires a VoLTE service with registration and voice call functionalities. He/she requires different service access points to expose these functionalities on different planes. One *SAP* should expose the registration and voice call on the control plane, and another *SAP* should expose the voice call on the data plane. According to the *NFR*s, the tenant requires 60 requests per second with the maximum request size of five units for the voice call functionality on the control plane. For the voice call on data plane, he/she requires 6000 units of throughput. For the registration on the control plane, he/she requires 50 requests per second with the maximum request size of three units. Figure 6.6 shows the *NSReq* model of our use case using Papyrus tool. In this *NSReq* model, we have simplified the *Exposed Functionality* elements to the associations between the *SAPR*s and the *FR*s.

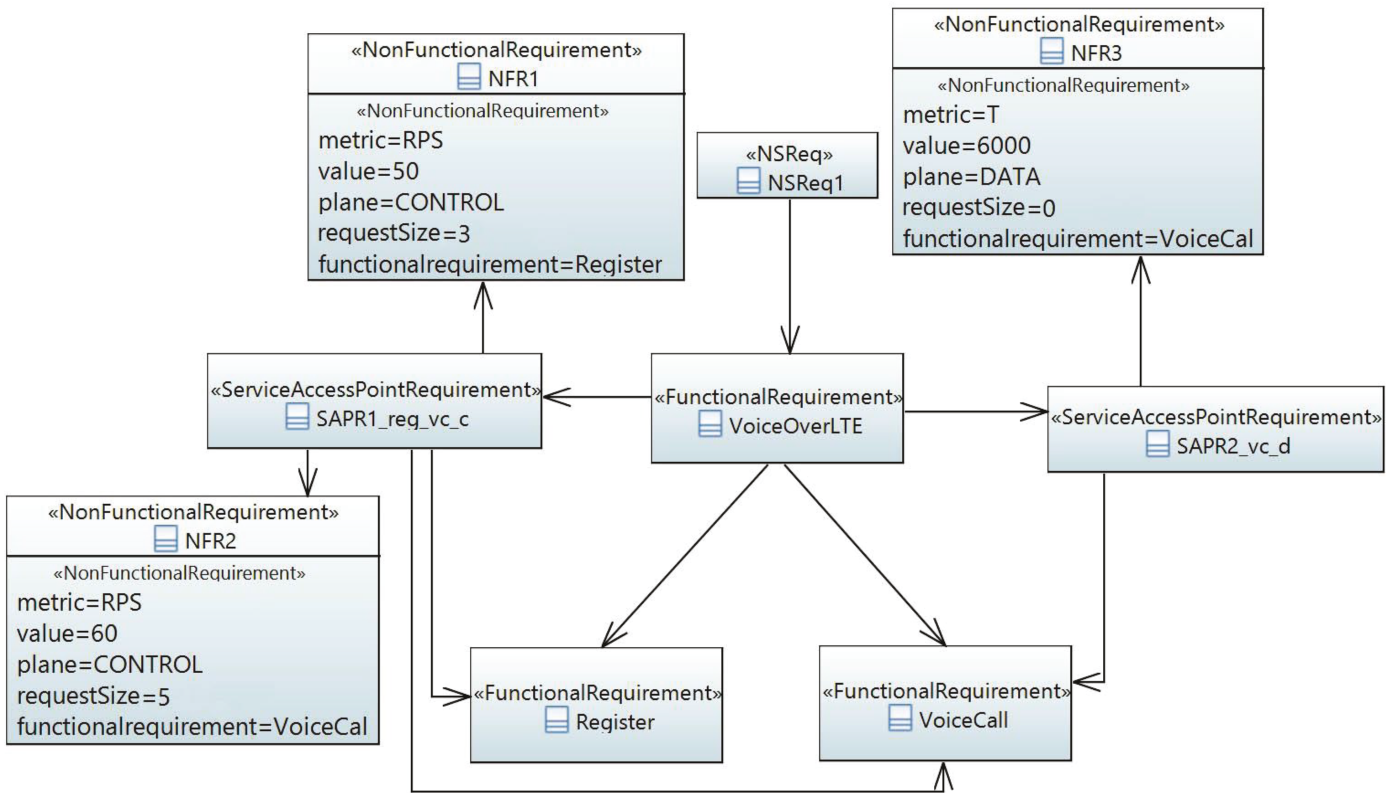

Figure 6.6 – The NSReq model for the case study

6.2.2 NF Ontology

The *NFO* model in our case study contains high level functionalities including VoLTE, IPTV, Online gaming, etc. However, our focus is on the VoLTE functionality. In this model, the VoLTE is composed of five lower level functionalities. Among them, voice call, user info storage, authentication, and registration are mandatory, and messaging is optional. Voice call and messaging functionalities depend on authentication. These two dependencies have the context of VoLTE functionality, i.e. these dependencies exist in the VoLTE decomposition. Authentication depends on registration and user info storage, and registration depends on user info storage. All these dependencies have no context, i.e. they exist regardless of the decomposition they are in. Figure 6.7 shows a portion of our case study *NFO* model in Papyrus tool that focuses on the VoLTE functionality. The numbers shown on the realization relations refer to the same relations in the other portion of the *NFO* model in Figure 6.8.

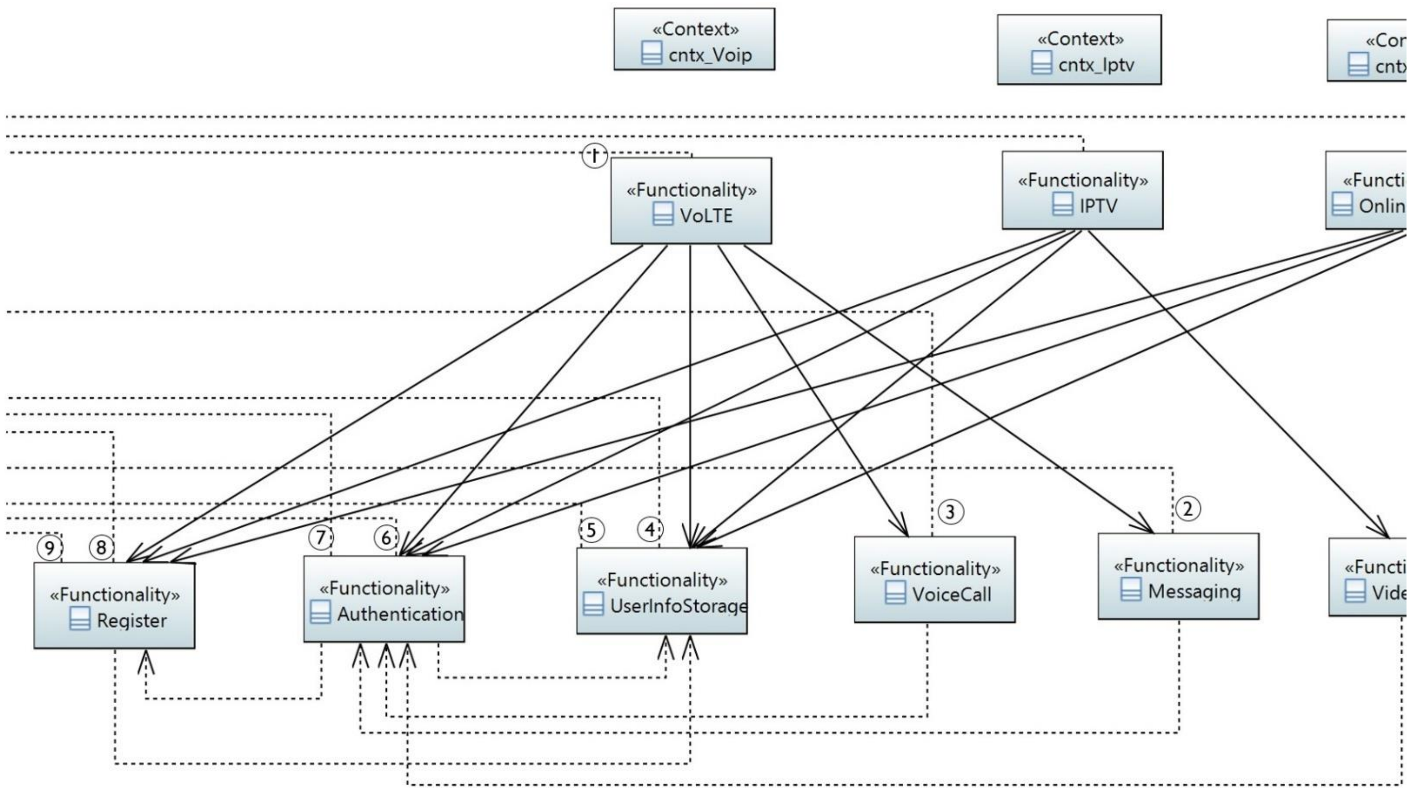

Figure 6.7 – The NFO model for the case study (functional portion)

On the architectural side of the *NFO* model, we have the IMS architectural block with two compositions, as shown in Figure 6.8. The one on the right is the typical IMS composition which is a simplified version of the standard IMS [13]. It is composed of AS, HSS, P-CSCF, I-CSCF, and S-CSCF as introduced in the Background.

The AS realizes the messaging and voice call functionalities, and it exposes the IMS standard interfaces including ISC and Mb [13]. The MRFP module exposes the Mb interface according to [13], but for simplification, the AS exposes this interface in our *NFO*. The ISC interface exposes the voice call and messaging functionalities on the control plane, and the Mb exposes them on the data plane. The HSS realizes the user info storage functionality. It exposes this functionality on the control plane through Cx interface. The S-CSCF realizes the authentication and registration functionalities and exposes them on the control plane through the Mw interface.

It exposes the ISC interface to communicate with the AS for these functionalities. It also exposes the Cx interface to communicate with the HSS in order to store and retrieve the user information. I-CSCF and P-CSCF also expose the Mw interface to communicate with each other and S-CSCF for the registration functionality. P-CSCF exposes the Gm as a service access point for the IMS, and the users connect to it for registration and requesting voice calls.

According to the architectural dependencies shown in Figure 6.8, S-CSCF communicates with AS for the voice call functionality. It communicates with HSS for user info storage. It also communicates with I-CSCF and P-CSCF for registration and voice call. I-CSCF communicates with the P-CSCF for registration. P-CSCF associates with one SAP for receiving the registration and voice call requests on the control plane through its Gm interface. AS also associates with one SAP for the voice call on the data plane through its Mb interface. The users connect to this SAP to exchange their voice call content. Table 6.1 shows the dependencies in the typical IMS composition and the details of their *ADep Interfaces* elements.

The IMS composition on the left side of Figure 6.8 is called merged IMS, as the CSCFs are merged into one architectural block called Core IMS. The other architectural blocks in this composition are IMS Locator, DB, and AS. It is a simplified version of the merged IMS architecture proposed in [46]. The Core IMS architectural block realizes the registration and authorization functionalities. IMS Locator manages the assignment of the users to Core IMS instances and routing their requests to their corresponding Core IMS. The DB is a proprietary database for user info storage. AS is the same architectural block as discussed in the previous composition with the same functionalities and interfaces. Core IMS exposes two proprietary interfaces which we refer to as X1 and X2. X1 is for communication with the IMS Locator regarding the registration and voice call functionalities. X2 is for communicating with the DB regarding the user info storage. The DB exposes the X2 and Cx interfaces, to communicate with the Core IMS and

also the S-CSCF. The IMS Locator exposes IMS standard interfaces including the Gm and the ISC in addition to the X1 interface.

According to the architectural dependencies shown in Figure 6.8, the IMS Locator communicates with the Core IMS for registration and voice call on the control plane. It also communicates with the AS for voice call on the control plane. Core IMS communicates with the DB for the user info storage. A SAP is associated with the IMS Locator through its Gm interface, and it exposes the registration and voice call on the control plane.

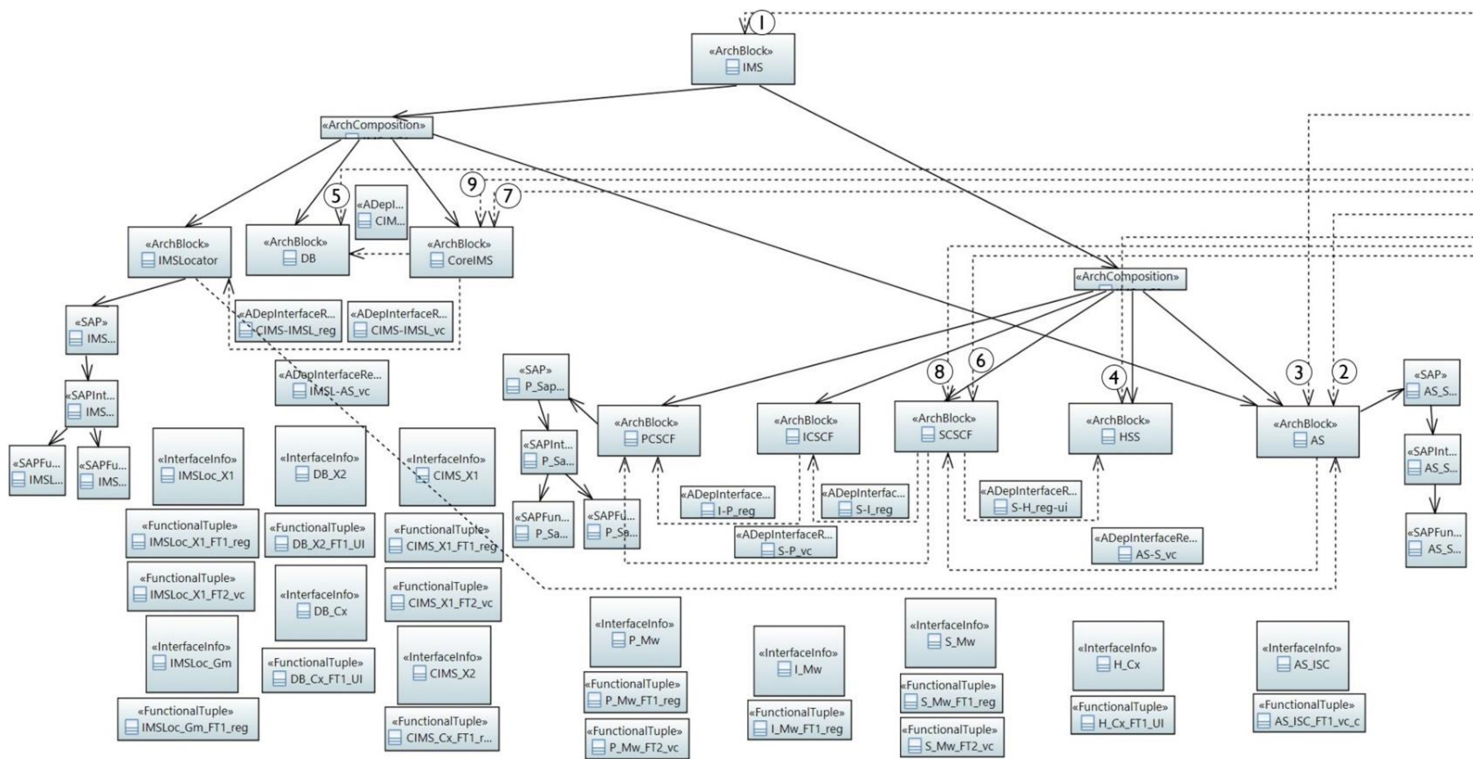

Figure 6.8 – The NFO model in the case study (architectural portion)

ADep Interfaces Dependencies	No.	Plane	Source Interface	Source Func.	Source Roles	Source Protocol	Target Interface	Target Func.	Target Roles	Target Protocol
I-CSCF-P-CSCF	1	Control	ICSCF: Mw	Registration	Server, Input, Output	SIP	PCSCF: Mw	Registration	Client, Input, Output	SIP
S-CSCF-P-CSCF	2	Control	SCSCF: Mw	Voice Call	Server, Client, Input, Output	SIP	PCSCF: Mw	Voice Call	Server, Client, Input, Output	SIP
S-CSCF-I-CSCF	3	Control	SCSCF: Mw	Registration	Server, Input, Output	SIP	ICSCF: Mw	Registration	Client, Input, Output	SIP
S-CSCF-HSS	4	Control	SCSCF: Cx	Registration	Client, Input, Output	Diameter	HSS: Cx	User Info Storage	Server, Input, Output	Diameter
AS-S-CSCF	5	Control	AS: ISC	Voice Call	Server, Input, Output	SIP	SCSCF: ISC	Voice Call	Client, Input, Output	SIP

Table 6.1 - The details of the architectural dependencies in the NFO

6.2.3 VNF Catalog

Our *VNF Catalog* model contains the *VNF Package Info* elements of all the VNFs that our *NFO* model is based on. These VNFs include the AS, HSS, S-CSCF, I-CSCF, P-CSCF, Core IMS, IMS Locator, and DB. Figure 6.9 shows our *VNF Catalog* model in Papyrus tool.

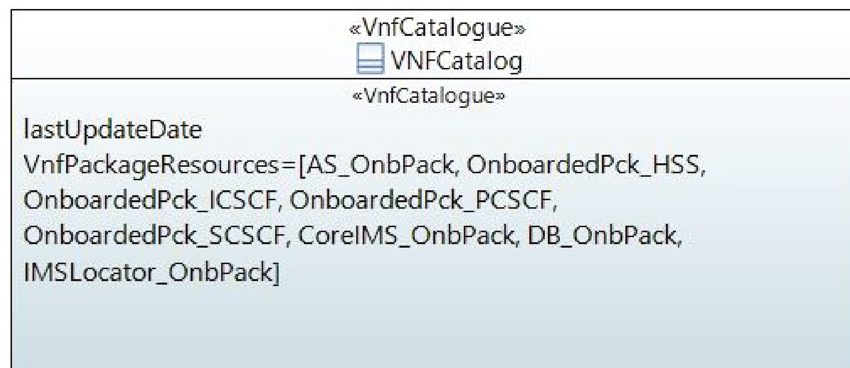

Figure 6.9 – The VNF Catalog model in the case study

6.2.3.1 *VNFD*

In our approach, the most important information of a VNF is the information from its *VNFAD* and *VnfDf*s. As a result, all of our *VNFD*s have a simple and similar structure. Each

VNF has only one *Vdu*, except the AS which has two *Vdu*s, one for the data, and one for the control plane. In each VNF, there is a *VduCpd* and a *VnfExtCpd* element for each *VNF Interface*. For simplification, we have considered the bitrate requirement of each *VduCpd* same as the throughput of the VNF interface it exposes. Each VNF has two *VnfDf*s that each has two *Instantiation Level*s. The number of instances of the *Vdu* in the *Instantiation Level*s from the smallest to the largest is 1 to 4 respectively. Figure 6.10 shows the P-CSCF *VNFD* in Papyrus tool as an example of our *VNFD*s.

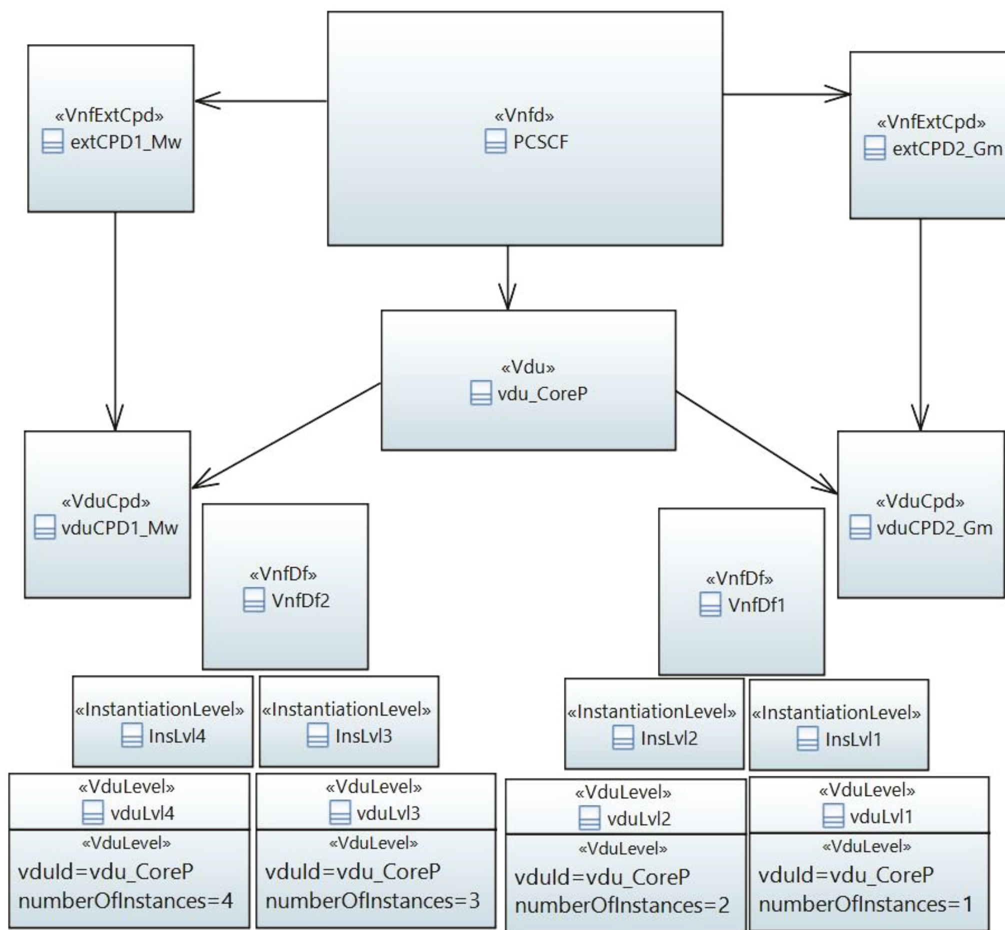

Figure 6.10 – The P-CSCF VNFD model for the case study

6.2.3.2 *VNFAD*

In our case study, each *VNFAD* has one *VNF Interface* element for each of the interfaces of its corresponding VNF, as discussed in Section 6.2.3.1. Each *VNFAD* has one or multiple *Flow*

Transformation elements as well. Figure 6.11 shows the P-CSCF *VNFAD* in Papyrus tool as an example of our *VNFAD*s. Table 6.2 shows the details of the *VNF Interface* elements of the VNFs in the typical IMS composition. Table 6.3 shows the details of the *Flow Transformation* elements of these VNFs. The results of the *Propagation Flow* design and dimensioning the VNFs in Transformation 4 and 5 in our case study is according to the information in these tables.

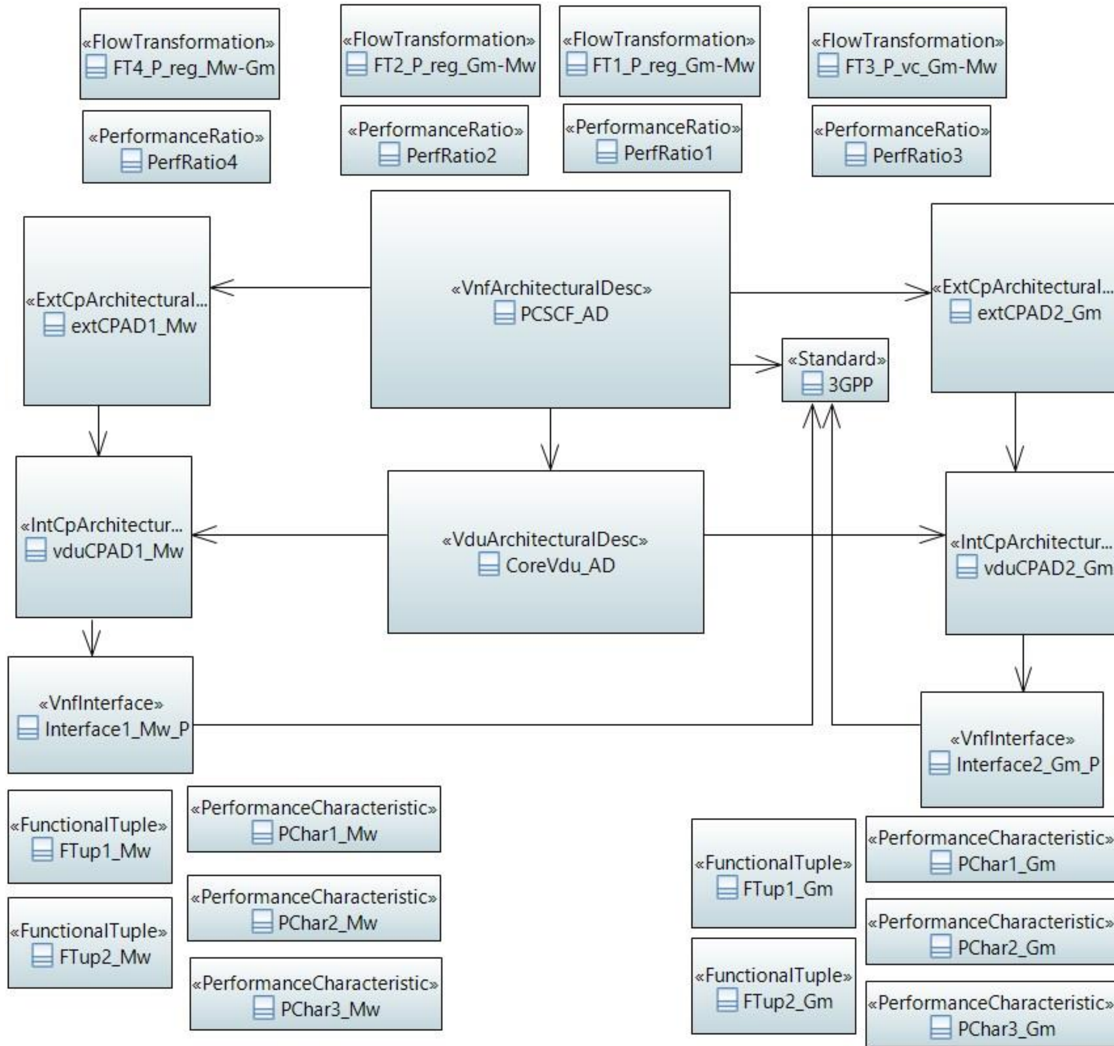

Figure 6.11 – The P-CSCF VNFAD model for the case study

VNF	Interface	Functional Characteristic			QoS Characteristic	
		Functionality	Plane	Roles	Metric	Value
P-CSCF	Mw	Registration	Control	Server, Input, Output	Throu.	100
		Voice Call	Control	Server, Client Input, Output	RPS	30
	Gm	Registration	Control	Client, Input, Output	Throu.	100
		Voice Call	Control	Server, Client Input, Output	RPS	30
I-CSCF	Mw	Registration	Control	Server, Client Input, Output	Throu.	100
					RPS	20
S-CSCF	Mw	Registration	Control	Server, Client Input, Output	Throu.	300
		Voice Call	Control	Server, Client Input, Output	RPS	40
	Cx	Registration	Control	Client, Input, Output	Throu.	600
					RPS	50
	ISC	Voice Call	Control	Client, Output	Throu.	300
					RPS	50
HSS	Cx	User Info Storage	Control	Server, Input, Output	Throu.	250
					RPS	70
AS	ISC	Voice Call	Control	Server, Client Input, Output	Throu.	200
					RPS	20
	Mb	Voice Call	Data	Input, Output	Throu.	5000

Table 6.2 - The VNF Interface elements of the VNFs in the typical IMS composition

VNF	Flow Transformation							QoS Ratio	
	No.	Plane	Source->Target Interface	Source Func.	Source Role	Target Func.	Target Role	Source->Target Metric	Ratio
P-CSCF	1	Control	Gm -> Mw	Registration	Server	Registration	Client	RPS -> RPS	1
	2	Control	Gm -> Mw	Registration	Input	Registration	Output	Thro->Thro	1
	3	Control	Gm -> Mw	Voice Call	Server	Voice Call	Client	Thro->Thro	1
	4	Control	Mw -> Gm	Registration	Input	Registration	Output	Thro->Thro	1
I-CSCF	5	Control	Mw -> Mw	Registration	Server	Registration	Client	RPS -> RPS	1
	6	Control	Mw -> Mw	Registration	Input	Registration	Output	Thro->Thro	1
S-CSCF	7	Control	Mw -> ISC	Voice Call	Server	Voice Call	Client	Thro->Thro	1
	8	Control	Mw -> Cx	Registration	Server	Registration	Client	RPS -> RPS	2
	9	Control	Mw -> Cx	Registration	Input	Registration	Output	Thro -> Thro	2
	10	Control	Mw -> Mw	Registration	Server	Registration	Output	RPS->Thro	0.1
HSS	11	Control	Cx -> Cx	User Info Storage	Server	User Info Storage	Output	RPS -> Thro	3
	12	Control	Cx -> Cx	User Info Storage	Input	User Info Storage	Output	Thro -> Thro	1
AS	13	Data	Mb -> Mb	Voice Call	Input	Voice Call	Output	Thro->Thro	1

Table 6.3 - The Flow Transformation elements of the VNFs in the typical IMS composition

6.2.3.3 *VNF Package Info*

In our *VNF Package Info* models, we have defined only the *VNF Package Info* and its *User Defined Data* elements, as these are the only elements in our scope. In the former element, we reference the *VNFD* element, and in the latter, we reference the *VNFAD* element. Figure 6.12 shows the P-CSCF *VNF Package Info* as an example.

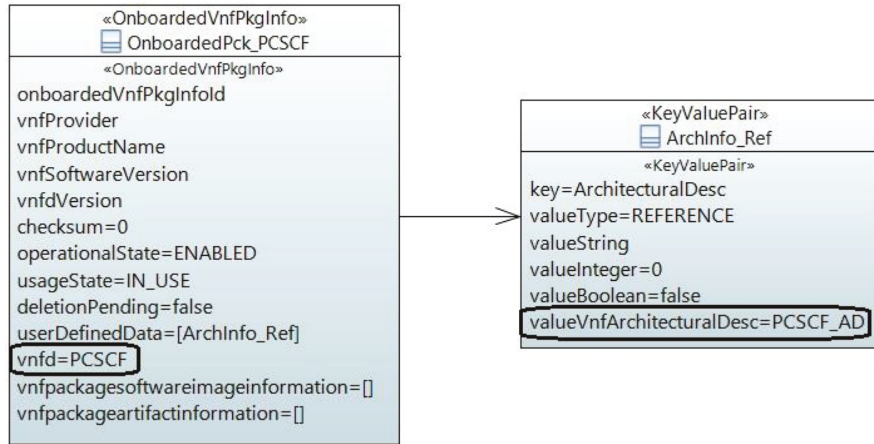

Figure 6.12 – The P-CSCF VNF Package Info model in the case study

6.2.4 Protocol Stack

Figure 6.13 shows our *Protocol Stack* model in Papyrus tool.

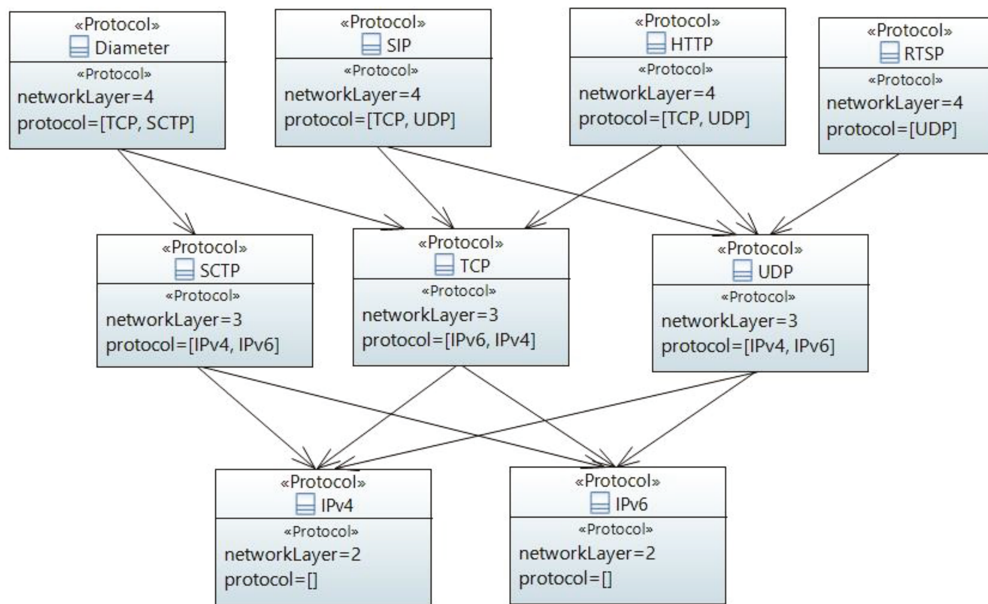

Figure 6.13 – The Protocol Stack model in the case study

6.2.5 Results of the NSD generation and refinement process

In this section, we will discuss the results of each transformation in our case study. The output of all the transformations is UML files that contain the desired output models.

6.2.5.1 *Transformation 1 (Initializing the SM model)*

The input of the Transformation 1 is the *NSReq* model discussed in Section 6.2.1. The result of this transformation is an *SM* model that contains all the *NSReq* elements transformed into corresponding *SM* elements, according to Section 4.2.1. We refer to it as *SM1*. *SM1* contains the VoLTE *Functionality* element decomposed to voice call and registration *Functionality* elements. It also has all the *SAPR*s and the *NFR*s of the *NSReq* model. Figure 6.14 shows *SM1* in the UML file resulted from Transformation 1.

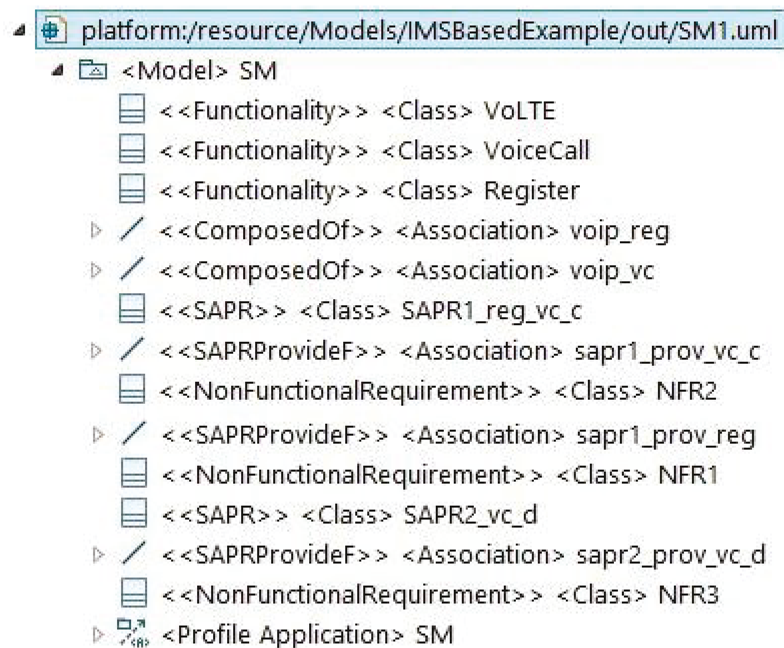

Figure 6.14 – The SM1 generated from the Transformation 1

6.2.5.2 *Transformation 2 (Decomposing the SM model)*

The inputs of the Transformation 2 are the *NFO* model, as discussed in Section 6.2.2, and the *SM1*. This transformation results in a new *SM* model that we refer to as *SM2*. It contains the VoLTE, registration, and voice call functionalities from *SM1*. The authentication and the user info storage functionalities are in the VoLTE core decomposition in the *NFO*. Therefore, they have been added to the VoLTE decomposition in *SM2*. The dependencies of the aforementioned

functionalities also have been added to the *SM2* from the *NFO*. The Functionality elements like IPTV, messaging, and video streaming have not been added as they are not relevant to the requirements.

In the *NFO*, all the architectural blocks in both of the IMS compositions realize at least one functionality that is in *SM2*. Therefore, the IMS architectural block and both of its compositions with all of their elements have been added to *SM2*. These elements include all the architectural blocks, their dependencies and *ADep Interfaces* elements, and their *Interface*s and *SAP*s as shown in Figure 6.7. Figure 6.15 shows a portion of *SM2* in the UML file resulted from Transformation 2.

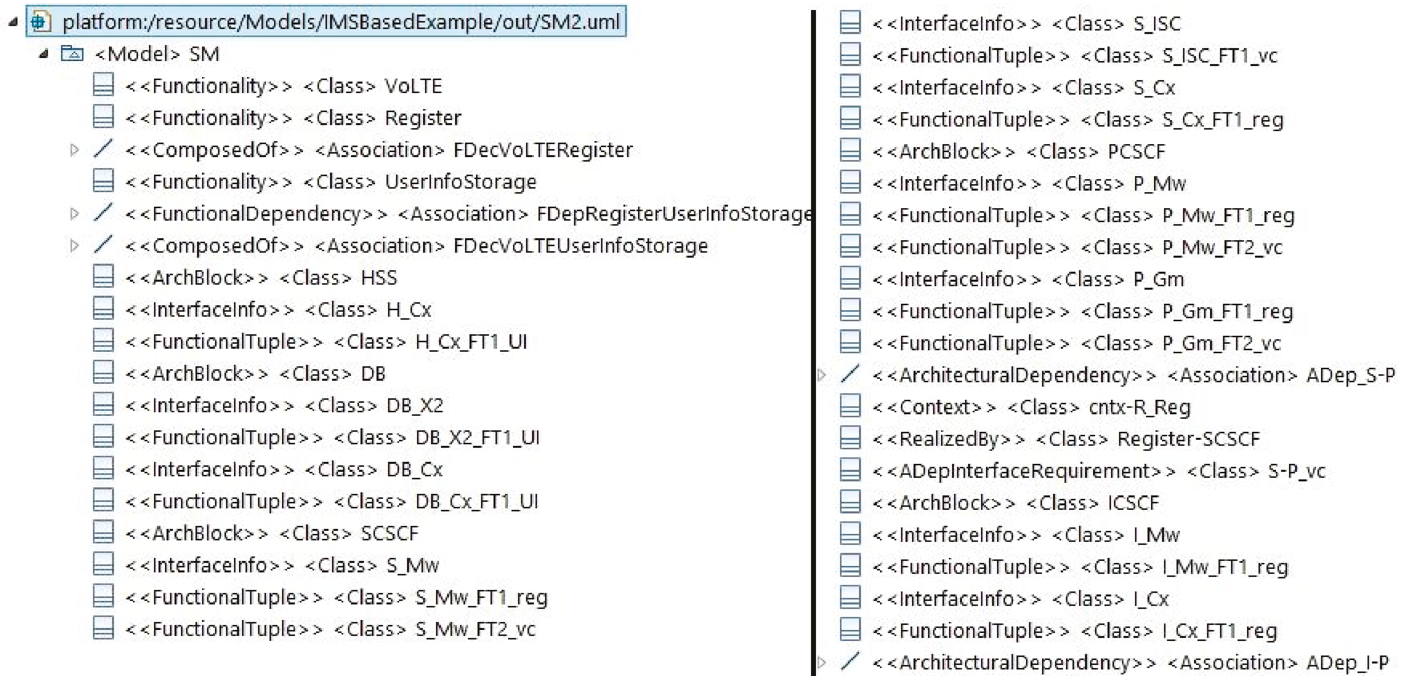

Figure 6.15 - A portion of the SM2 generated from the Transformation 2

6.2.5.3 *Transformation 3 (Selecting the VNFs and Generating the FGs)*

The input models of the Transformation 3 are the *VNF Catalog*, as discussed in Section 6.2.3, and the *SM2*. This transformation results in an *SM* model that we refer to as *SM3*.

Each VNF in the *VNF Catalog* matches with an architectural block in the *SM3*. Therefore, all the VNFs have been added into the *SM3* from the *VNF Catalog* in this step.

Potentially, we can design two *FFG*s from the *SM3*. One is composed of the VoLTE functionality, and the other is composed of the functionalities in the VoLTE's decomposition. No VNF realizes the IMS architectural block in the *SM3*, thus, no VNF realizes the VoLTE functionality either. Therefore, the first *FFG* has been dismissed. From the second *FFG*, eight *AFG*s have been generated. They are based on all the combinations of the architectural blocks in the *SM3* that realize the *FFG*'s functionalities. Table 4 shows the generated *AFG*s and the mapping between their architectural blocks and the *FFG*'s functionalities.

AFGs \ FFG's Functionalities	Registration	Authentication	User Info Storage	Voice Call
1	P-CSCF I-CSCF S-CSCF	P-CSCF I-CSCF S-CSCF	HSS	AS
2	P-CSCF I-CSCF S-CSCF	Core IMS IMS Locator	HSS	AS
3	P-CSCF I-CSCF S-CSCF	P-CSCF I-CSCF S-CSCF	DB	AS
4	P-CSCF I-CSCF S-CSCF	Core IMS IMS Locator	DB	AS
5	Core IMS IMS Locator	P-CSCF I-CSCF S-CSCF	HSS	AS
6	Core IMS IMS Locator	Core IMS IMS Locator	HSS	AS
7	Core IMS IMS Locator	P-CSCF I-CSCF S-CSCF	DB	AS
8	Core IMS IMS Locator	Core IMS IMS Locator	DB	AS

Table 6.4 – The AFGs generated in Transformation 3

All the generated *AFG*s except the *AFG1* and *8* are incomplete, as their architectural blocks are from different IMS compositions. The P/I/S-CSCF and the HSS have no interface that matches the interfaces of the IMS Locator and the Core IMS. Among the incomplete *AFG*s, the *AFG2*, *4*, *5*, *6*, and *7* have architectural blocks from both of these groups. The *AFG Completion* procedure is not able to create the missing architectural dependencies for these *AFG*s due to the lack of matching interfaces. Therefore, they remain incomplete. All the architectural blocks in the *AFG3* are from the typical IMS composition except the DB. DB exposes the Cx interface through which it can communicate with the S-CSCF. Therefore, the *AFG Completion* procedure is able to create the missing dependency between the DB and the S-CSCF. The complete *AFG*s generated in this transformation are *AFG1*, *3*, and *8*.

One *Pre-VNFFG* has been generated for each of these *AFG*s since there is a one-to-one mapping between the VNFs and the architectural blocks in the *SM3*. Figure 6.16 shows a portion of the *SM3* in the UML file resulted from Transformation 3. This portion is related to the generated FGs.

<<FFG>> <Class> FFG 1
<<AFG>> <Class> AFG 1-FFG 1
<<ABFTie>> <Class> AB-F-Tie AFG 1-FFG 1 SCSCF-Register
<<ABFTie>> <Class> AB-F-Tie AFG 1-FFG 1 HSS-UserInfoStorage
<<ABFTie>> <Class> AB-F-Tie AFG 1-FFG 1 SCSCF-Authentication
<<ABFTie>> <Class> AB-F-Tie AFG 1-FFG 1 AS-VoiceCall
<<ACFFGTie>> <Class> AC-F-Tie AFG 1-FFG 1-IMS-AC2
<<ACFFGTie>> <Class> AC-F-Tie AFG 1-FFG 1-IMS-AC1
<<AFG>> <Class> AFG 2-FFG 1
<<ABFTie>> <Class> AB-F-Tie AFG 2-FFG 1 SCSCF-Register
<<ABFTie>> <Class> AB-F-Tie AFG 2-FFG 1 HSS-UserInfoStorage
<<ABFTie>> <Class> AB-F-Tie AFG 2-FFG 1 CoreIMS-Authentication
<<ABFTie>> <Class> AB-F-Tie AFG 2-FFG 1 AS-VoiceCall
<<ACFFGTie>> <Class> AC-F-Tie AFG 2-FFG 1-IMS-AC2
<<ACFFGTie>> <Class> AC-F-Tie AFG 2-FFG 1-IMS-AC1
<<AFG>> <Class> AFG 3-FFG 1
<<ABFTie>> <Class> AB-F-Tie AFG 3-FFG 1 SCSCF-Register
<<ABFTie>> <Class> AB-F-Tie AFG 3-FFG 1 DB-UserInfoStorage
<<ABFTie>> <Class> AB-F-Tie AFG 3-FFG 1 SCSCF-Authentication
<<ABFTie>> <Class> AB-F-Tie AFG 3-FFG 1 AS-VoiceCall
<<ACFFGTie>> <Class> AC-F-Tie AFG 3-FFG 1-IMS-AC2
<<ACFFGTie>> <Class> AC-F-Tie AFG 3-FFG 1-IMS-AC1
<<AFG>> <Class> AFG 4-FFG 1
<<ABFTie>> <Class> AB-F-Tie AFG 4-FFG 1 SCSCF-Register
<<ABFTie>> <Class> AB-F-Tie AFG 4-FFG 1 DB-UserInfoStorage
<<ABFTie>> <Class> AB-F-Tie AFG 4-FFG 1 CoreIMS-Authentication
<<ABFTie>> <Class> AB-F-Tie AFG 4-FFG 1 AS-VoiceCall
<<ACFFGTie>> <Class> AC-F-Tie AFG 4-FFG 1-IMS-AC2
<<ACFFGTie>> <Class> AC-F-Tie AFG 4-FFG 1-IMS-AC1
<<AFG>> <Class> AFG 5-FFG 1
<<ABFTie>> <Class> AB-F-Tie AFG 5-FFG 1 CoreIMS-Register
<<ABFTie>> <Class> AB-F-Tie AFG 5-FFG 1 HSS-UserInfoStorage
<<ABFTie>> <Class> AB-F-Tie AFG 5-FFG 1 SCSCF-Authentication
<<ABFTie>> <Class> AB-F-Tie AFG 5-FFG 1 AS-VoiceCall
<<ACFFGTie>> <Class> AC-F-Tie AFG 5-FFG 1-IMS-AC1
<<ACFFGTie>> <Class> AC-F-Tie AFG 5-FFG 1-IMS-AC2
<<AFG>> <Class> AFG 6-FFG 1
<<ABFTie>> <Class> AB-F-Tie AFG 6-FFG 1 CoreIMS-Register
<<ABFTie>> <Class> AB-F-Tie AFG 6-FFG 1 HSS-UserInfoStorage
<<ABFTie>> <Class> AB-F-Tie AFG 6-FFG 1 CoreIMS-Authentication
<<ABFTie>> <Class> AB-F-Tie AFG 6-FFG 1 AS-VoiceCall
<<ACFFGTie>> <Class> AC-F-Tie AFG 6-FFG 1-IMS-AC1
<<ACFFGTie>> <Class> AC-F-Tie AFG 6-FFG 1-IMS-AC2
<<AFG>> <Class> AFG 7-FFG 1
<<ABFTie>> <Class> AB-F-Tie AFG 7-FFG 1 CoreIMS-Register
<<ABFTie>> <Class> AB-F-Tie AFG 7-FFG 1 DB-UserInfoStorage
<<ABFTie>> <Class> AB-F-Tie AFG 7-FFG 1 SCSCF-Authentication
<<ABFTie>> <Class> AB-F-Tie AFG 7-FFG 1 AS-VoiceCall
<<ACFFGTie>> <Class> AC-F-Tie AFG 7-FFG 1-IMS-AC1
<<ACFFGTie>> <Class> AC-F-Tie AFG 7-FFG 1-IMS-AC2
<<AFG>> <Class> AFG 8-FFG 1
<<PreVNFFG>> <Class> PreVNFFG 1-AFG 1-FFG 1
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 1-FFG 1 SCSCF-SCSCF
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 1-FFG 1 HSS-HSS
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 1-FFG 1 AS-AS
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 1-FFG 1 PCSCF-PCSCF
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 1-FFG 1 ICSCF-ICSCF
<<PreVNFFG>> <Class> PreVNFFG 1-AFG 3-FFG 1
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 3-FFG 1 SCSCF-SCSCF
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 3-FFG 1 DB-DB
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 3-FFG 1 AS-AS
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 3-FFG 1 PCSCF-PCSCF
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 3-FFG 1 ICSCF-ICSCF
<<PreVNFFG>> <Class> PreVNFFG 1-AFG 8-FFG 1
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 8-FFG 1 CoreIMS-CoreIMS
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 8-FFG 1 DB-DB
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 8-FFG 1 AS-AS
<<VNFDABTie>> <Class> VNFD-AB-Tie PreVNFFG 1-AFG 8-FFG 1 IMSLocator-IMSLocator

Figure 6.16 – The FGs portion of the SM3 generated in Transformation 3

6.2.5.4 *Transformation 4 (Generating the generic NSDs)*

The input models of the Transformation 4 are the *SM3* and the *Protocol Stack* model, as discussed in Section 6.2.4. This transformation results in two UML files. One contains the generated generic *NSD*s, and the other is an *SM* model that we refer to as *SM4*. For each of the *Pre-*

*VNFFG*s in the *SM3* one generic *NSD* model has been generated. Figure 6.17 shows the UML file containing these three *NSD* models.

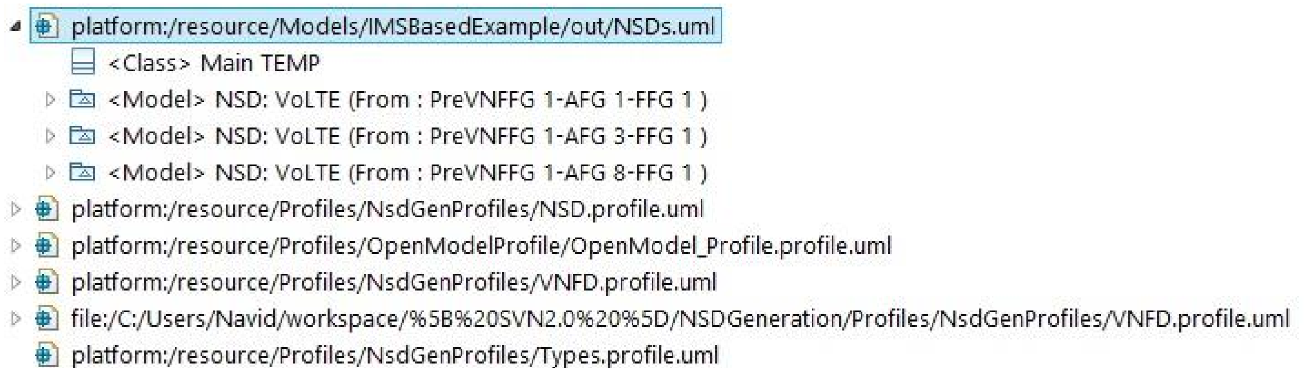

Figure 6.17 – The NSD models generated in Transformation 4

As an example, Figure 6.18 shows the *NSD1* model that is originated from the *AFG1* according to the typical IMS composition. According to the architectural dependencies in the typical IMS, three *VLD*s have been generated. They exist at the top of the figure along with their *Connectivity Type* elements. One of the *VLD*s is an *E-LAN* type that connects the P/I/S-CSCF VNFs through their Mw interfaces. The other *VLD*s are *E-Line* type, and they connect the S-CSCF VNF to the HSS and the AS VNFs through their Cx and ISC interfaces, respectively.

Two *VNFFGD*s have been created, one for the data and the other for the control plane of the network service. In the control plane, *VNFFGD* two *Propagation Flow*s have been designed, one for registration and the other for voice call functionalities on the control plane. These two functionalities are exposed by the SAP associated with the Gm interface of P-CSCF. *P_Sap-Gm* is the *SAPD* of this SAP as shown in Figure 6.18. In the registration *flow*, P-CSCF receives the registration requests from the users and sends them to the I-CSCF. I-CSCF finds the assigned S-CSCF to each user and routes the requests to these S-CSCFs. Each S-CSCF inquires the information of the users from the HSS for the authentication. After S-CSCF performs the registration functionality, it sends an acknowledgment message to the user through the incoming

route of the request. The voice call *flow* on the control plane in our case study is a simplified version of the session setup procedure in the IMS [13]. In this *flow*, the users send the voice call requests to P-CSCF. P-CSCF routes the requests to the S-CSCFs assigned to the users. Each S-CSCF sends a request to the AS to setup up a connection for each requested call.

In the data plane *VNFFGD* one *Propagation Flow* has been designed for the voice call functionality on the data plane. This functionality is exposed by the SAP associated with the Mb interface of the AS. *AS_Sap-Mb* is the *SAPD* of this SAP as shown in Figure 6.18. In this *flow*, each user in the call sends the voice call content through this SAP to the Mb interface. AS sends the content to the other user through the same interface.

One *NFPD* have been generated according to each of the aforementioned *flow*s. These *NFPD*s include *NFPD Register-CONTROL*, *NFPD VoiceCall-CONTROL*, and *NFPD VoiceCall-DATA* as shown in Figure 6.18.

<Model> NSD: VoLTE (From : PreVNFFG 1-AFG 1-FFG 1)

- <<Nsd>> <Class> NSD: VoLTE (From : PreVNFFG 1-AFG 1-FFG 1)
- <<ConnectivityType>> <Class> MESH-SIP
- <<NsVirtualLinkDesc>> <Class> Mw_Sequence {'ADep_S-P', 'ADep_S-I', 'ADep_I-P'}
- <<ConnectivityType>> <Class> LINE-Diameter
- <<NsVirtualLinkDesc>> <Class> Cx_Sequence {'ADep_S-HSS'}
- <<ConnectivityType>> <Class> LINE-SIP
- <<NsVirtualLinkDesc>> <Class> ISC_Sequence {'ADep_AS-S'}
- <<Sapd>> <Class> P_Sap-Gm
- <<Sapd>> <Class> AS_Sap-Mb
- <<Vnffgd>> <Class> Vnffgd DATA plane
- <<CpdPool>> <Class> CpPool-Sapds
- <<Nfpd>> <Class> NFPD VoiceCall-DATA
- <<Vnffgd>> <Class> Vnffgd CONTROL plane
- <<CpdPool>> <Class> CpPool-ICSCF
- <<CpdPool>> <Class> CpPool-SCSCF
- <<CpdPool>> <Class> CpPool-PCSCF
- <<CpdPool>> <Class> CpPool-HSS
- <<CpdPool>> <Class> CpPool-AS
- <<CpdPool>> <Class> CpPool-Sapds
- <<Nfpd>> <Class> NFPD VoiceCall-CONTROL
- <<Nfpd>> <Class> NFPD Register-CONTROL
- <<NsDf>> <Class> NsDf - NsDf - NSD: VoLTE (From : PreVNFFG 1-AFG 1-FFG 1)
- <<VirtualLinkProfile>> <Class> VlProfile - Mw_Sequence {'ADep_S-P', 'ADep_S-I', 'ADep_I-P'}
- <<VirtualLinkProfile>> <Class> VlProfile - Cx_Sequence {'ADep_S-HSS'}
- <<VirtualLinkProfile>> <Class> VlProfile - ISC_Sequence {'ADep_AS-S'}
- <<VnfProfile>> <Class> VnfProfile - SCSCF
- <<NsVirtualLinkConnectivity>> <Class> NSVLConn-VlProfile - Mw_Sequence {'ADep_S-P', 'ADep_S-I', 'ADep_I-P'}-SCSCF
- <<NsVirtualLinkConnectivity>> <Class> NSVLConn-VlProfile - Cx_Sequence {'ADep_S-HSS'}-SCSCF
- <<NsVirtualLinkConnectivity>> <Class> NSVLConn-VlProfile - ISC_Sequence {'ADep_AS-S'}-SCSCF
- <<VnfProfile>> <Class> VnfProfile - HSS
- <<NsVirtualLinkConnectivity>> <Class> NSVLConn-VlProfile - Cx_Sequence {'ADep_S-HSS'}-HSS
- <<VnfProfile>> <Class> VnfProfile - AS
- <<NsVirtualLinkConnectivity>> <Class> NSVLConn-VlProfile - ISC_Sequence {'ADep_AS-S'}-AS
- <<VnfProfile>> <Class> VnfProfile - PCSCF
- <<NsVirtualLinkConnectivity>> <Class> NSVLConn-VlProfile - Mw_Sequence {'ADep_S-P', 'ADep_S-I', 'ADep_I-P'}-PCSCF
- <<VnfProfile>> <Class> VnfProfile - ICSCF
- <<NsVirtualLinkConnectivity>> <Class> NSVLConn-VlProfile - Mw_Sequence {'ADep_S-P', 'ADep_S-I', 'ADep_I-P'}-ICSCF

Figure 6.18 – The NSD model for the typical IMS composition form Transformation 4

Figure 6.19 shows the aforementioned *flow*s generated in the *SM4*.

<<PropagationFlow>> <Class> Propagation Flow VoiceCall-DATA
<<SmVnfInterface>> <Class> SmInt Interface2_Mb: 1
<<SmVnfInterface>> <Class> SmInt Interface2_Mb: 2
<<PropagationFlow>> <Class> Propagation Flow VoiceCall-CONTROL
<<SmVnfInterface>> <Class> SmInt Interface2_Gm_P: 1
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_P: 2
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_S: 3
<<SmVnfInterface>> <Class> SmInt Interface3_ISC_S: 4
<<SmVnfInterface>> <Class> SmInt Interface3_ISC_vc: 5
<<PropagationFlow>> <Class> Propagation Flow Register-CONTROL
<<SmVnfInterface>> <Class> SmInt Interface2_Gm_P: 1
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_P: 2
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_I: 3
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_I: 4
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_S: 5
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_S: 6
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_I: 7
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_I: 8
<<SmVnfInterface>> <Class> SmInt Interface1_Mw_P: 9
<<SmVnfInterface>> <Class> SmInt Interface2_Gm_P: 10
<<SmVnfInterface>> <Class> SmInt Interface2_Cx_S: 6
<<SmVnfInterface>> <Class> SmInt Interface1_Cx_H: 7
<<SmVnfInterface>> <Class> SmInt Interface1_Cx_H: 8
<<SmVnfInterface>> <Class> SmInt Interface2_Cx_S: 9

Figure 6.19 - The Propagation Flows in the SM4 from Transformation 4

Table 6.5 shows the details of each *Propagation Flow*, i.e. the sequence of their interfaces, and the interfaces' characteristics related to the *flow*s. It also shows the source of finding each interface in the *flow*, i.e. the *Flow Transformation* or the *ADep Interfaces* elements.

Interface Sequence / Propagation Flow	1 (Starting Int.)	2 (exit)	3 (entry)	4 (exit)	5 (entry)
	6 (exit)	7 (entry)	8 (exit)	9 (entry)	10 (exit)
	6 (exit)	7 (entry)	8 (exit)	9 (entry)	10 (exit)
Registration - Control Plane	P-CSCF: Gm	P-CSCF: Mw	I-CSCF: Mw	I-CSCF: Mw	S-CSCF: Mw
	Func.: Registra. **Plane:** Control **Roles:** Server, Input	**Func.:** Registra. **Plane:** Control **Roles:** Client, Output **Flow Tran.:** 1,2	**Func.:** Registra. **Plane:** Control **Roles:** Server, Input **ADep. Int.:** 1	**Func.:** Registra. **Plane:** Control **Roles:** Client, Output **Flow Tran.:** 5,6	**Func.:** Registra. **Plane:** Control **Roles:** Server, Input **ADep. Int.:** 3
	S-CSCF: Mw	I-CSCF: Mw	I-CSCF: Mw	P-CSCF: Mw	P-CSCF: Gm
	Func.: Registra. **Plane:** Control **Roles:** Output **Flow Tran.:** 10	**Func.:** Registra. **Plane:** Control **Roles:** Input **ADep. Int.:** 3	**Func.:** Registra. **Plane:** Control **Roles:** Output **Flow Tran.:** 5,6	**Func.:** Registra. **Plane:** Control **Roles:** Input **ADep. Int.:** 1	**Func.:** Registra. **Plane:** Control **Roles:** Output **Flow Tran.:** 4
	S-CSCF: Cx	HSS: Cx	HSS: Cx	S-CSCF: Cx	
	Func.: Registra. **Plane:** Control **Roles:** Client, Output **Flow Tran.:** 8.9	**Func.:** UIStorag **Plane:** Control **Roles:** Server, Input **ADep. Int.:** 4	**Func.:** UIStorag **Plane:** Control **Roles:** Output **Flow Tran.:** 11, 12	**Func.:** Registra. **Plane:** Control **Roles:** Input **ADep. Int.:** 4	
Voice Call - Control Plane	P-CSCF: Gm	P-CSCF: Mw	S-CSCF: Mw	S-CSCF: ISC	AS: ISC
	Func.: Voice C. **Plane:** Control **Roles:** Server, Input	**Func.:** Voice C. **Plane:** Control **Roles:** Client, Output **Flow Tran.:** 3	**Func.:** Voice C. **Plane:** Control **Roles:** Server, Input **ADep. Int.:** 2	**Func.:** Voice C. **Plane:** Control **Roles:** Client, Output **Flow Tran.:** 7	**Func.:** Voice C. **Plane:** Control **Roles:** Server, Input **ADep. Int.:** 5
Voice Call - Data Plane	AS: Mb	AS: Mb			
	Func.: Voice C. **Plane:** Data **Roles:** Input	**Func.:** Voice C. **Plane:** Data **Roles:** Output **Flow Tran.:** 13			

Table 6.5 - Details of the Propagation Flows in the case study

6.2.5.5 *Transformation 5 (Updating the NFO)*

In our case study, no *NFO* update happens.

6.2.5.6 *Transformation 6 (Refining the generic NSDs)*

The input models of the Transformation 6 are the *SM4* and the UML file containing the *NSD* models. This transformation results in a UML file containing the refined *NSD* models, and an *SM* model that we refer to as *SM5*. The *NFR1* (50 RPS, max request size 3), as shown in Figure 6.6, relates to the first *flow* in Table 6.5, since both are for registration functionality on the control plane. *NFR2* (60 RPS, max request size 5) and *NFR3* (6000 Throughput), also, relate to the second and the third *flow* in Table 6.5. In this transformation, each of these *NFR*s has been propagated through their related *flow*s.

Table 6.6 shows the details of propagating the *NFR*s in this case study. It shows the *QR*s calculated for each interfaces' appearance in the *flow*s and their metric.IU. The maximum metric.IU for each appearance is specified by a circle. According to Section 5.2.1.3, the total.IU of an interface is the summation of its maximum metric.IUs of each of its appearances in all the *flow*s. The total.IUs of the interfaces of each VNF in this case study are:

P-CSCF) Gm: 4.72, Mw: 4.72

I-CSCF) Mw: 5.05

S-CSCF) Mw: 2.26, Cx: 2.05, ISC: 1

HSS) Cx: 2.63

AS) ISC: 1.5, Mb: 2.4

As discussed in Section 5.2.2.1, the VNFC.RI for a VNFC is equal to the ceiling of the maximum among its interfaces' total.IUs. As discussed in Section 6.2.3.1, each VNF in this case study has only one VNFC, except for the AS. The VNFC.RI for these VNFCs are: P-CSCF-VNFC1: 5, I-CSCF-VNFC1: 6, S-CSCF-VNFC1: 3, HSS-VNFC1: 3, AS-VNFC1 (ISC interface): 2, AS-VNFC2 (Mb interface): 3.

Interface Sequence / Propagation Flow	1 (Starting Int.)	2 (exit)	3 (entry)	4 (exit)	5 (entry)
	6 (exit)	**7 (entry)**	**8 (exit)**	**9 (entry)**	**10 (exit)**
	6 (exit)	**7 (entry)**	**8 (exit)**	**9 (entry)**	**10 (exit)**
Registration - Control Plane	P-CSCF: Gm	P-CSCF: Mw	I-CSCF: Mw	I-CSCF: Mw	S-CSCF: Mw
	QR1: 50 RPS **metric.IU1:** 1.67 **QR2:** 150 Thro. **metric.IU2:** 1.5	**QR1:** 50 RPS (by Flow Trans. 1) **metric.IU1:** 1.67 **QR2:** 150 Throu. (by Flow Trans. 2) **metric.IU2:** 1.5	**QR1:** 50 RPS **metric.IU1:** 2.5 **QR2:** 150 Throu. **metric.IU2:** 1.5	**QR1:** 50 RPS (by Flow Trans. 5) **metric.IU1:** 2.5 **QR2:** 150 Throu. (by Flow Trans. 6) **metric.IU2:** 1.5	**QR1:** 50 RPS **metric.IU1:** 1.25 **QR2:** 150 Throu. **metric.IU2:** 0.5
	S-CSCF: Mw	I-CSCF: Mw	I-CSCF: Mw	P-CSCF: Mw	P-CSCF: Gm
	QR1: 5 Throu. (by Flow Tra. 10) **metric.IU1:** 0.016	**QR1:** 5 Throu. **metric.IU1:** 0.05	**QR1:** 5 Throu. (by Flow Tran. 6) **metric.IU1:** 0.05	**QR1:** 5 Throu. **metric.IU1:** 0.05	**QR1:** 5 Throu. (by Flow Trans. 4) **metric.IU1:** 0.05
	S-CSCF: Cx	HSS: Cx	HSS: Cx	S-CSCF: Cx	
	QR1: 100 RPS (by Flow Tran. 8) **metric.IU1:** 2 **QR2:** 300 Throu. (by Flow Tran. 9) **metric.IU2:** 0.5	**QR1:** 100 RPS **metric.IU1:** 1.43 **QR2:** 300 Throu. **metric.IU2:** 1.2	**QR1:** 300 Throu. (by Flow Tra. 11) **metric.IU1:** 1.2 **QR2:** 300 Throu. (by Flow Tra. 12) **metric.IU2:** 1.2	**QR1:** 300 Throu. **metric.IU1:** 0.5 **QR2:** 300 Throu. **metric.IU2:** 0.5	
Voice Call - Control Plane	P-CSCF: Gm	P-CSCF: Mw	S-CSCF: Mw	S-CSCF: ISC	AS: ISC
	QR1: 60 RPS **metric.IU1:** 2 **QR2:** 300 Throu. **metric.IU2:** 3	**QR1:** 300 Throu. (by Flow Trans. 3) **metric.IU1:** 3	**QR1:** 300 Throu. **metric.IU1:** 1	**QR1:** 300 Throu. (by Flow Trans. 7) **metric.IU1:** 1	**QR1:** 300 Throu. **metric.IU1:** 1.5
Voice Call - Data Plane	AS: Mb	AS: Mb			
	QR1: 6000 Thro. **metric.IU1:** 1.2	**QR1:** 6000 Thro. (by Flow Tran. 13) **metric.IU1:** 1.2			

Table 6.6 - Details of the NFR propagation in the case study

Table 6.7 shows the details of dimensioning the VNFs in this case study according to the VNFC.RIs. Each row shows the result of dimensioning each VNF for the specified *Instantiation Level*, i.e. it shows the $VNF.RI_{VNFC}$s and the $VNF.RI_{IL}$. For each VNF, the selected *Instantiation Level* for each *VnfDf*, i.e. with the minimum $VNF.RI_{IL}$ is specified with a dashed or a solid

circle. The final solution for dimensioning each VNF, i.e. with the minimum VNF.RI_{IL} is specified with a solid circle. Therefore, the required number of instances for the VNFs are P-CSCF: 2, I-CSCF: 2, S-CSCF: 1, HSS: 1, AS: 1.

VnfDfs \ VNFs		P-CSCF (VNF1 Dimensioning)				I-CSCF (VNF3 Dimensioning)				S-CSCF (VNF2 Dimensioning)			
		VNFC1 (RI=5)	*VNF.* RI_{VNFC}	VNF.RI	Total VNFCs	VNFC1 (RI=6)	*VNF.* RI_{VNFC}	VNF.RI	Total VNFCs	VNFC1 (RI=3)	*VNF.* RI_{VNFC}	VNF.RI	Total VNFCs
Df1	Ins. Lvl. 1	1	$\lceil 5/1 \rceil = 5$	**5**	5	1	$\lceil 6/1 \rceil = 6$	**6**	6	1	$\lceil 3/1 \rceil = 3$	**3**	3
Df1	Ins. Lvl. 2	2	$\lceil 5/2 \rceil = 3$	**3**	6	2	$\lceil 6/2 \rceil = 3$	**3**	6	2	$\lceil 3/2 \rceil = 2$	**2**	4
Df2	Ins. Lvl. 1	3	$\lceil 5/3 \rceil = 2$	**2**	6	3	$\lceil 6/3 \rceil = 2$	**2**	6	3	$\lceil 3/3 \rceil = 1$	**1**	3
Df2	Ins. Lvl. 2	4	$\lceil 5/4 \rceil = 2$	**2**	8	4	$\lceil 6/4 \rceil = 2$	**2**	8	4	$\lceil 3/4 \rceil = 1$	**1**	4

VnfDfs \ VNFs		HSS (VNF1 Dimensioning)				AS (VNF1 Dimensioning)					
		VNFC1 (RI=3)	*VNF.* RI_{VNFC}	VNF.RI	Total VNFCs	VNFC1 (RI=2)	*VNF.* RI_{VNFC}	VNFC2 (RI=3)	*VNF.* RI_{VNFC}	VNF.RI	Total VNFCs
Df1	Ins. Lvl. 1	1	$\lceil 3/1 \rceil = 3$	**3**	3	1	$\lceil 2/1 \rceil = 2$	1	$\lceil 3/1 \rceil = 3$	**3**	6
Df1	Ins. Lvl. 2	2	$\lceil 3/2 \rceil = 2$	**2**	4	2	$\lceil 2/2 \rceil = 2$	2	$\lceil 3/2 \rceil = 2$	**2**	8
Df2	Ins. Lvl. 1	3	$\lceil 3/3 \rceil = 1$	**1**	3	3	$\lceil 2/3 \rceil = 1$	3	$\lceil 3/3 \rceil = 1$	**1**	6
Df2	Ins. Lvl. 2	4	$\lceil 3/4 \rceil = 1$	**1**	4	4	$\lceil 2/4 \rceil = 1$	4	$\lceil 3/4 \rceil = 1$	**1**	8

Table 6.7 - Details of dimensioning the VNFs in the case study

All the information presented in Tables 6.6 and 6.7 has been stored in the *SM5*. The *VNF Profile*s in the refined *NSD* model also have been enriched accordingly. The minimum and the maximum root bitrate requirements for the *E-Line* virtual link between the S-CSCF and AS are both equal to 900, according to Equation 4. Similarly, these parameters for the *E-Line* virtual link between the S-CSCF and HSS are both equal to 1800. For the *E-LAN* virtual link between the P/I/S-CSCF VNFs, the minimum leaf and root bitrate requirements are equal to 900 and

1500 respectively, according to Equations 5 and 6. The maximum leaf and root bitrate requirements for this virtual link are also equal to 900 and 2100. The *NsVl Profile*s in the refined *NSD* model have been enriched according to these bitrate requirements. Based on dimensioning the VNFs, 4 *NS Level*s have been generated for the *NSD*. Figure 6.20 shows the last two generated *NS Level*s along with their *VnfToLevelMapping* and *NsVlToLevelMapping* elements.

<<NsLevel>> <Class> Instantiation Lvl 2
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - AS: 1
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - SCSCF: 1
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - HSS: 1
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - PCSCF: 2
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - ICSCF: 1
<<LinkBitrateRequirements>> <Class> BitrateReq-NS LEVEL for 2-1
<<VirtualLinkToLevelMapping>> <Class> VL to Level Mapping - ISC_Sequence {'ADep_AS-S'}: 2-1
<<LinkBitrateRequirements>> <Class> BitrateReq-NS LEVEL for 2-2
<<VirtualLinkToLevelMapping>> <Class> VL to Level Mapping - Mw_Sequence {'ADep_S-P', 'ADep_S-I', 'ADep_I-P'}: 2-2
<<LinkBitrateRequirements>> <Class> BitrateReq-NS LEVEL for 2-3
<<VirtualLinkToLevelMapping>> <Class> VL to Level Mapping - Cx_Sequence {'ADep_S-HSS'}: 2-3
<<NsLevel>> <Class> Instantiation Lvl 3
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - AS: 1
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - SCSCF: 1
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - HSS: 1
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - PCSCF: 2
<<VnfToLevelMapping>> <Class> Vnf to Level Mapping - ICSCF: 2
<<LinkBitrateRequirements>> <Class> BitrateReq-NS LEVEL for 3-1
<<VirtualLinkToLevelMapping>> <Class> VL to Level Mapping - ISC_Sequence {'ADep_AS-S'}: 3-1
<<LinkBitrateRequirements>> <Class> BitrateReq-NS LEVEL for 3-2
<<VirtualLinkToLevelMapping>> <Class> VL to Level Mapping - Mw_Sequence {'ADep_S-P', 'ADep_S-I', 'ADep_I-P'}: 3-2
<<LinkBitrateRequirements>> <Class> BitrateReq-NS LEVEL for 3-3
<<VirtualLinkToLevelMapping>> <Class> VL to Level Mapping - Cx_Sequence {'ADep_S-HSS'}: 3-3

Figure 6.20 – A portion of the NsDf in the refined NSD from Transformation 6

6.3 Discussion

In this chapter, we presented the prototype tool for deriving the generic *NSD*s from an *NSReq* and refining these *NSD*s according to the *NFR*s. We have developed this prototype in order to demonstrate the feasibility and the application of our method to real case studies, e.g. VoLTE using IMS architecture [13].

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we proposed a model-driven approach to automatically design network services and generate network service descriptors from high level network service requirements. These requirements define at a high level of abstraction the required network services from the functional, architectural, and QoS perspectives. The generated network service descriptors can be used by the NFV-MANO to deploy and manage the network services. Our approach mainly consists of two parts.

In the first part, we design network services according to the functional and architectural requirements. Our method decomposes the high level requirements using an ontology. Accordingly, it selects the suitable VNFs among the VNFs available in a catalog. The method defines the flows and the forwarding graphs of the network services according to the dependencies between the VNFs. Based on the forwarding graphs, it generates an NSD model complying with the ETSI NFV standards [9, 10] for each network service in the solution set.

In the second part, we refine the generated NSDs according to the NFRs in the network service requirements. Therefore, these NSDs will be able to provide the range of QoS specified

by the NFRs. This method propagates the NFRs through the flows of each network service and the VNFs involved in the flows. It selects the suitable deployment flavor of each VNF and dimensions it accordingly. It also calculates the required capacity for the virtual links connecting the VNFs according to the VNFs dimensioning. Finally, the method tailors the deployment flavor of each NSD accordingly.

During the NS design and NSD generation process, we also enrich automatically the ontology. Taking into account new information provided in the network service requirements, our method adds new functionalities and decompositions into the ontology. It may also add new aliases to the existing functionalities. Note that the ontology may also be enriched based on new VNFs added to the catalog.

As proof of concept, we developed a prototype tool and illustrated our approach with the VoLTE service.

The NFV framework enables the automation of activities in network service orchestration and management, e.g. on-boarding, deployment, lifecycle management, etc. All these activities depend and come after network service design. Nowadays, experts in the OSS/BSS divisions design network services manually. Our work fills the gap and fits very well in the big picture of network automation. It is a step towards zero-touch in Telecommunications.

7.2 Potential Future Work

There are a few directions for further investigations in this research.

- The configuration of the application aspect of a network service is required before its instantiation, e.g. the configuration required for a Firewall according to the policies, or configuring a VoIP server to whether use SIP or H.323 protocol. This requires information about the application aspect of the network service and it is out of the NFV scope. To complement

this thesis, one can devise a method to automatically generate such a configuration for the network service.

- Our method does not check the consistency of the network service requirements, and we assume the given requirements are consistent. A method to check the consistency of the network service requirements can prevent generating faulty NSDs due to inconsistent requirements.
- Full validation of our method using real and industry level case studies is desirable.

Bibliography

[1] "Network Functions Virtualisation (NFV); Architectural Framework: ETSI GS NFV 002 V1.2.1," 12 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf.

[2] "Network Functions Virtualisation (NFV); Virtual Network Functions Architecture: ETSI GS NFV-SWA 001 V1.1.1," 12 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-SWA/001_099/001/01.01.01_60/gs_NFV-SWA001v010101p.pdf.

[3] "Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV: ETSI GS NFV 003 V1.4.1," 08 2018. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/003/01.04.01_60/gs_NFV003v010401p.pdf.

[4] M.Pearce, S.Zeadally and R.Hunt, "Virtualization: Issues, Security Threats, and Solutions," *ACM Computing Society,* vol. 45, no. 2, 2013.

[5] "European Telecommunications Standards Institute (ETSI)," [Online]. Available: https://www.etsi.org/.

[6] "Network Function Virtualization (NFV)," ETSI, [Online]. Available: https://www.etsi.org/technologies/nfv.

[7] "Network Functions Virtualisation (NFV); Use Cases: ETSI GS NFV 001," 10 2013. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf.

[8] "Network Functions Virtualisation (NFV); Management and Orchestration: ETSI GS NFV-MAN 001 V1.1.1," 12 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf.

[9] "Network Functions Virtualisation (NFV) Release 2; Management and Orchestration; Network Service Templates Specification: ETSI GS NFV-IFA 014 V2.4.1," 02 2018. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/014/02.04.01_60/gs_NFV-IFA014v020401p.pdf.

[10] "Network Functions Virtualisation (NFV) Release 2; Management and Orchestration; Report on NFV Information Model: ETSI GR NFV-IFA 015 V2.4.1," 02 2018. [Online]. Available: http://www.etsi.org/deliver/etsi_gr/NFV-IFA/001_099/015/03.01.01_60/gr_NFV-IFA015v030101p0.zip.

[11] M. Abbasipour, M. Sackmann, F. Khendek and M. Toeroe, "A Model-Based Approach for User Requirements Decomposition and Component Selection," *Formalisms for Reuse and Systems Integration,* pp. 173-202, 2015.

[12] J. Hyun, J. Li, C. Im, J.-H. Yoo and J. W.-K. Hong, "A VoLTE Traffic Classification Method in LTE Network," in *The 16th Asia-Pacific Network Operations and Management Symposium*, Hsinchu, 2014.

[13] "3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; IP Multimedia Subsystem (IMS); Stage 2 (Release 14): 3GPP TS 23.228 V14.4.0," June 2017. [Online]. Available: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=821.

[14] "Network Functions Virtualisation (NFV) Release 2; Management and Orchestration; VNF Descriptor and Packaging Specification: ETSI GS NFV-IFA 011 V2.4.1," 02 2018. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/011/02.04.01_60/gs_NFV-IFA011v020401p.pdf.

[15] H. Wang and S. Rupp, "Skype VoIP service- architecture and comparison," in *INFOTECH Seminar Advanced Communication Services (ACS)*, 2005.

[16] "MDA: Model Driven Architecture," Object Management Group (OMG), [Online]. Available: https://www.omg.org/mda/. [Accessed 26 February 2019].

[17] S. Mustafiz, N. Nazarzadeoghaz, G. Dupont, F. Khendek and M. Toeroe, "A Model-Driven Process Enactment Approach for Network Service Design," in *International Conference on System Design Languages* , Budapest, 2017.

[18] "Cellular Standards for 3G: ITU's IMT-2000 Family," International Telecommunication Union (ITU), [Online]. Available: https://www.itu.int/osg/spu/imt-2000/technology.html#Cellular%20Standards%20for%20the%20Third%20Generation.

[19] G. Camarillo and M. A. Garcia-Martin, The 3G IP multimedia subsystem (IMS): Merging the Internet and the Cellular Worlds, 2nd ed., West Sussex: Wiley, 2006.

[20] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach, Pearson, 2012.

[21] "Third Generation Partnership Project (3GPP)," [Online]. Available: http://www.3gpp.org/.

[22] J. Rosenberg, H. Schulzrinne, G. Camarillo and A. Johnston, "SIP: Session Initiation Protocol," Internet Engineering Task Force (IETF), RFC 3261, June 2002. [Online]. Available: https://tools.ietf.org/html/rfc3261.

[23] M. Poikselkä, G. Mayer, H. Khartabil and A. Niemi, The IMS: IP Multimedia Concepts and Services in Mobile Domain, West Sussex: Wiley, 2004.

[24] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," Internet Engineering Task Force (IETF), RFC 3550, July 2003. [Online]. Available: https://tools.ietf.org/html/rfc3550.

[25] O. Levin, "H.323 Uniform Resource Locator (URL) Scheme Registration," Internet Engineering Task Force (IETF), RFC 3508, April 2003. [Online]. Available: https://tools.ietf.org/html/rfc3508#ref-3.

[26] 3GPP, "Long Term Evolution (LTE)," [Online]. Available: http://www.3gpp.org/technologies/keywords-acronyms/98-lte.

[27] "The Evolved Packet Core," 3GPP, [Online]. Available: http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core.

[28] "Ethernet Services Definitions - Phase II - MEF 6.1," April 2008. [Online]. Available: https://mef.net/PDF_Documents/technical-specifications/MEF6-1.pdf.

[29] M. Brambilla, J. Cabot and M. Wimmer, Model-Driven Software Engineering In Practice, 2nd ed., Morgan & Claypool, 2017.

[30] "Object Management Group (OMG)," [Online]. Available: https://www.omg.org/.

[31] T. Stahl and M. Völter, Model-Driven Software Development: Technology, Engineering, Management, West Sussex: Wiley, 2006.

[32] "Unified Modeling Language (OMG UML) Infrastructure, version 2.4.1," Object Management Group (OMG), 2011. [Online]. Available: https://www.omg.org/spec/UML/2.4.1/About-UML/.

[33] "OMG Unified Modeling language (OMG UML) Specification 2.4, Superstructure," 2011. [Online]. Available: https://www.omg.org/spec/UML/2.4.1/About-UML/.

[34] "Papyrus Modeling Environment," 08 March 2017. [Online]. Available: http://download.eclipse.org/modeling/mdt/papyrus/updates/releases/neon/.

[35] M. Abbasipour, "A Framework for Requirements Decomposition, SLA Management and Dynamic System Reconfiguration," PhD Thesis, Concordia University, 2018.

[36] "ATL/User Guide-The ATL Language," [Online]. Available: https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language. [Accessed 11 July 2018].

[37] "Object Constraint Language (OCL), version 2.4," Object Management Group (OMG), Febdruary 2014. [Online]. Available: https://www.omg.org/spec/OCL/.

[38] "Service Availability Forum (SA Forum)," [Online]. Available: http://www.saforum.org/.

[39] Z. Oster, G. Santhanam and S. Basu, "Decomposing the Service Composition Problem," in *8th IEEE European Conference on Web Services*, Ayia Napa, 2010.

[40] Z. Oster, G. Santhanam and S. Basu, "Identifying Optimal Composite Services by Decomposing the Service Composition Problem," in *IEEE International Conference on Web Services*, Washington, DC, 2011.

[41] S. Sun and J. Zhao, "A decomposition-based approach for service composition with global QoS guarantees," *Information Sciences,* vol. 199, pp. 138-153, 2012.

[42] Y. Liu, L. Wu and S. Liu, "A Novel QoS-Aware Service Composition Approach Based on Path Decomposition," in *IEEE Asia-Pacific Services Computing Conference*, Guilin, 2012.

[43] C. Bartsch, L. Shwartz, C. Ward, G. Grabarnik and M. J. Buco, "Decomposition of IT service processes and alternative service identification using ontologies," in *IEEE Network Operations and Management Symposium*, Salvador, Bahia, 2008.

[44] S. I. Kim and H. S. Kim, "Semantic Ontology-Based NFV Service Modeling," in *10th International Conference on Ubiquitous and Future Networks* , Prague, 2018.

[45] S. Sahhaf, W. Tavernier, D. Colle and M. Pickavet, "Network service chaining with efficient network function mapping based on service decompositions," in *1st IEEE Conference on Network Softwarization (NetSoft)*, London, 2015.

[46] G. Carella, M. Corici, P. Crosta, P. Comi and T. M. Bohnert, "Cloudified IP Multimedia Subsystem (IMS) for Network Function Virtualization (NFV)-based architectures," in *IEEE Symposium on Computers and Communications (ISCC)*, Funchal, 2014.

[47] D. Benavides, S. Segura and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems,* vol. 35, no. 6, pp. 615-636, 2010.

[48] A. Leon-Garcia and I. Widjaja, Communication Networks: Fundamental Concepts and Key Architectures, McGraw-Hill, 2004.

[49] R. Diestel, "The Basics," in *Graph Theory, 5th ed.*, Springer, 2016, pp. 1-35.